EBook Gratis

APRENDIZAJE Python Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#python

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: python-language

It is an unofficial and free Python Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Python Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Python Language

Observaciones

Python es un lenguaje de programación muy utilizado. Es:

- **Alto nivel** : Python automatiza las operaciones de bajo nivel, como la administración de memoria. Deja al programador con un poco menos de control, pero tiene muchos beneficios que incluyen la legibilidad del código y expresiones de código mínimas.
- **Propósito general** : Python está diseñado para ser utilizado en todos los contextos y entornos. Un ejemplo de un lenguaje de uso no general es PHP: está diseñado específicamente como un lenguaje de script de desarrollo web del lado del servidor. En contraste, Python *puede* usarse para el desarrollo web del lado del servidor, pero también para crear aplicaciones de escritorio.
- **Escrito dinámicamente** : cada variable en Python puede hacer referencia a cualquier tipo de datos. Una sola expresión puede evaluar datos de diferentes tipos en diferentes momentos. Debido a eso, el siguiente código es posible:

```
if something:
    x = 1
else:
    x = 'this is a string'
print(x)
```

- **Se escribe con fuerza** : durante la ejecución del programa, no se le permite hacer nada que sea incompatible con el tipo de datos con los que está trabajando. Por ejemplo, no hay conversiones ocultas de cadenas a números; una cadena hecha de dígitos nunca se tratará como un número a menos que la conviertas explícitamente:

```
1 + '1'  # raises an error
1 + int('1')  # results with 2
```

- **Amigable para principiantes :)** : la sintaxis y la estructura de Python son muy intuitivas. Es de alto nivel y proporciona construcciones destinadas a permitir la escritura de programas claros tanto a pequeña como a gran escala. Python es compatible con múltiples paradigmas de programación, incluidos la programación orientada a objetos, la imperativa y funcional o los estilos de procedimiento. Tiene una biblioteca estándar grande y completa y muchas bibliotecas de terceros fáciles de instalar.

Sus principios de diseño se describen en *el Zen de Python* .

Actualmente, hay dos ramas principales de lanzamiento de Python que tienen algunas diferencias

significativas. Python 2.x es la versión heredada, aunque sigue teniendo un uso generalizado. Python 3.x realiza un conjunto de cambios incompatibles con versiones anteriores que tienen como objetivo reducir la duplicación de características. Para obtener ayuda para decidir qué versión es la mejor para usted, consulte este artículo .

La documentación oficial de Python también es un recurso completo y útil, que contiene documentación para todas las versiones de Python, así como tutoriales para ayudarlo a comenzar.

Existe una implementación oficial del lenguaje suministrado por Python.org, generalmente denominado CPython, y varias implementaciones alternativas del lenguaje en otras plataformas de tiempo de ejecución. Estos incluyen IronPython (que ejecuta Python en la plataforma .NET), Jython (en el tiempo de ejecución de Java) y PyPy (que implementa Python en un subconjunto de sí mismo).

Versiones

Python 3.x

Versión	Fecha de lanzamiento
[3.7]	2017-05-08
3.6	2016-12-23
3.5	2015-09-13
3.4	2014-03-17
3.3	2012-09-29
3.2	2011-02-20
3.1	2009-06-26
3.0	2008-12-03

Python 2.x

Versión	Fecha de lanzamiento
2.7	2010-07-03
2.6	2008-10-02
2.5	2006-09-19

Versión	Fecha de lanzamiento
2.4	2004-11-30
2.3	2003-07-29
2.2	2001-12-21
2.1	2001-04-15
2.0	2000-10-16

Examples

Empezando

Python es un lenguaje de programación de alto nivel ampliamente utilizado para la programación de propósito general, creado por Guido van Rossum y lanzado por primera vez en 1991. Python cuenta con un sistema de tipo dinámico y gestión automática de memoria y soporta múltiples paradigmas de programación, incluyendo imperativo, orientado a objetos. Programación funcional, y estilos procesales. Tiene una biblioteca estándar grande y completa.

Dos versiones principales de Python están actualmente en uso activo:

- Python 3.x es la versión actual y está en desarrollo activo.
- Python 2.x es la versión heredada y solo recibirá actualizaciones de seguridad hasta 2020. No se implementarán nuevas funciones. Tenga en cuenta que muchos proyectos siguen utilizando Python 2, aunque la migración a Python 3 es cada vez más sencilla.

Puede descargar e instalar cualquiera de las versiones de Python aquí . Ver Python 3 contra Python 2 para una comparación entre ellos. Además, algunos terceros ofrecen versiones reenvasadas de Python que agregan bibliotecas de uso común y otras características para facilitar la configuración de casos de uso comunes, como matemáticas, análisis de datos o uso científico. Vea la lista en el sitio oficial .

Verificar si Python está instalado

Para confirmar que Python se instaló correctamente, puede verificarlo ejecutando el siguiente comando en su terminal favorita (si está usando el sistema operativo Windows, debe agregar la ruta de acceso de python a la variable de entorno antes de usarlo en el símbolo del sistema):

```
$ python --version
```

Python 3.x 3.0

Si tiene *Python 3* instalado y es su versión predeterminada (consulte la sección **Solución** de **problemas** para obtener más detalles), debería ver algo como esto:

```
$ python --version
Python 3.6.0
```

Python 2.x 2.7

Si tiene instalado *Python 2* y es su versión predeterminada (consulte la sección **Solución** de **problemas** para obtener más detalles), debería ver algo como esto:

```
$ python --version
Python 2.7.13
```

Si ha instalado Python 3, pero `$ python --version` genera una versión de Python 2, también tiene Python 2 instalado. Este es a menudo el caso de MacOS y muchas distribuciones de Linux. Use `$ python3` en `$ python3` lugar para usar explícitamente el intérprete de Python 3.

Hola, Mundo en Python usando IDLE

IDLE es un editor simple para Python, que viene incluido con Python.

Cómo crear el programa Hello, World en IDLE.

- Abra IDLE en su sistema de elección.
  - En versiones anteriores de Windows, se puede encontrar en `All Programs` en el menú de Windows.
  - En Windows 8+, busque `IDLE` o encuéntrelo en las aplicaciones que están presentes en su sistema.
  - En sistemas basados en Unix (incluyendo Mac), puede abrirlo desde el shell escribiendo `$ idle python_file.py` .
- Se abrirá una concha con opciones a lo largo de la parte superior.

En la cáscara, hay un indicador de tres corchetes de ángulo recto:

```
>>>
```

Ahora escriba el siguiente código en el indicador:

```
>>> print("Hello, World")
```

Presiona `Enter` .

```
>>> print("Hello, World")
Hello, World
```

Hola archivo de World Python

Crea un nuevo archivo `hello.py` que contenga la siguiente línea:

Python 3.x 3.0

```
print('Hello, World')
```

Python 2.x 2.6

Puede usar la función de `print` Python 3 en Python 2 con la siguiente declaración de `import` :

```
from __future__ import print_function
```

Python 2 tiene una serie de funcionalidades que pueden importarse opcionalmente desde Python 3 usando el módulo `__future__` , como se explica aquí .

Python 2.x 2.7

Si usa Python 2, también puede escribir la siguiente línea. Tenga en cuenta que esto no es válido en Python 3 y, por lo tanto, no se recomienda porque reduce la compatibilidad de código entre versiones.

```
print 'Hello, World'
```

En su terminal, navegue al directorio que contiene el archivo `hello.py` .

Escriba `python hello.py` , luego `python hello.py` la tecla `Intro` .

```
$ python hello.py
Hello, World
```

Deberías ver `Hello, World` impreso en la consola.

También puede sustituir `hello.py` con la ruta a su archivo. Por ejemplo, si tiene el archivo en su directorio de inicio y su usuario es "usuario" en Linux, puede escribir `python /home/user/hello.py` .

Ejecutar un shell interactivo de Python

Al ejecutar (ejecutar) el comando `python` en su terminal, se le presenta un shell interactivo de Python. Esto también se conoce como el intérprete de Python o REPL (para 'Leer Evaluar Imprimir Loop').

```
$ python
Python 2.7.12 (default, Jun 28 2016, 08:46:01)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello, World'
Hello, World
>>>
```

Si desea ejecutar Python 3 desde su terminal, ejecute el comando `python3` .

```
$ python3
Python 3.6.0 (default, Jan 13 2017, 00:00:00)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello, World')
Hello, World
>>>
```

Alternativamente, inicie la solicitud interactiva y cargue el archivo con `python -i <file.py>` .

En la línea de comando, ejecute:

```
$ python -i hello.py
"Hello World"
>>>
```

Hay varias formas de cerrar el shell de Python:

```
>>> exit()
```

o

```
>>> quit()
```

Alternativamente, `CTRL + D` cerrará el shell y lo pondrá nuevamente en la línea de comando de su terminal.

Si desea cancelar un comando que está escribiendo y volver a un indicador de comandos limpio, mientras permanece dentro del intérprete de intérprete, use `CTRL + C`.

Pruebe un shell interactivo de Python en línea .

Otras conchas en línea

Varios sitios web proporcionan acceso en línea a las conchas de Python.

Los depósitos en línea pueden ser útiles para los siguientes propósitos:

- Ejecute un pequeño fragmento de código desde una máquina que carece de la instalación de Python (teléfonos inteligentes, tabletas, etc.).
- Aprende o enseña Python básico.
- Resolver problemas de jueces en línea.

Ejemplos:

> Descargo de responsabilidad: los autores de la documentación no están afiliados a los recursos que se enumeran a continuación.

- https://www.python.org/shell/ - El shell de Python en línea alojado en el sitio web oficial de

Python.

- https://ideone.com/ : se utiliza ampliamente en la red para ilustrar el comportamiento del fragmento de código.
- https://repl.it/languages/python3 - Compilador en línea, IDE e intérprete en línea potente y simple. Codifique, compile y ejecute el código en Python.
- https://www.tutorialspoint.com/execute_python_online.php : shell UNIX con todas las funciones y un explorador de proyectos fácil de usar.
- http://rextester.com/l/python3_online_compiler - IDE simple y fácil de usar que muestra el tiempo de ejecución

Ejecutar comandos como una cadena

Python puede pasar un código arbitrario como una cadena en el shell:

```
$ python -c 'print("Hello, World")'
Hello, World
```

Esto puede ser útil cuando se concatenan los resultados de los scripts juntos en el shell.

Conchas y mas alla

Administración de paquetes : la herramienta recomendada por PyPA para instalar paquetes de Python es PIP . Para instalar, en su línea de comando ejecute `pip install <the package name>` . Por ejemplo, `pip install numpy` . (Nota: en Windows debe agregar pip a sus variables de entorno PATH. Para evitar esto, use `python -m pip install <the package name>`)

Shells : hasta ahora, hemos discutido diferentes formas de ejecutar código usando el shell interactivo nativo de Python. Los shells utilizan el poder interpretativo de Python para experimentar con el código en tiempo real. Los shells alternativos incluyen IDLE , una GUI pre-empaquetada, IPython , conocida por extender la experiencia interactiva, etc.

Programas : para el almacenamiento a largo plazo, puede guardar el contenido en archivos .py y editarlos / ejecutarlos como secuencias de comandos o programas con herramientas externas, como shell, IDE (como PyCharm), cuadernos Jupyter , etc. Los usuarios intermedios pueden usar estas herramientas; sin embargo, los métodos discutidos aquí son suficientes para comenzar.

El tutor de Python te permite recorrer el código de Python para que puedas visualizar cómo fluirá el programa, y te ayuda a entender dónde salió mal tu programa.

PEP8 define las pautas para formatear el código Python. Formatear bien el código es importante para que pueda leer rápidamente lo que hace el código.

Creando variables y asignando valores.

Para crear una variable en Python, todo lo que necesita hacer es especificar el nombre de la

variable y luego asignarle un valor.

```
<variable name> = <value>
```

Python usa = para asignar valores a las variables. No es necesario declarar una variable por adelantado (o asignarle un tipo de datos), al asignar un valor a una variable, se declara e inicializa la variable con ese valor. No hay forma de declarar una variable sin asignarle un valor inicial.

```
# Integer
a = 2
print(a)
# Output: 2

# Integer
b = 9223372036854775807
print(b)
# Output: 9223372036854775807

# Floating point
pi = 3.14
print(pi)
# Output: 3.14

# String
c = 'A'
print(c)
# Output: A

# String
name = 'John Doe'
print(name)
# Output: John Doe

# Boolean
q = True
print(q)
# Output: True

# Empty value or null data type
x = None
print(x)
# Output: None
```

La asignación de variables funciona de izquierda a derecha. Así que lo siguiente te dará un error de sintaxis.

```
0 = x
=> Output: SyntaxError: can't assign to literal
```

No puede utilizar palabras clave de python como un nombre de variable válido. Puedes ver la lista de palabras clave por:

```
import keyword
print(keyword.kwlist)
```

Reglas para nombrar variables:

1. Los nombres de las variables deben comenzar con una letra o un guión bajo.

```
 x  = True   # valid
 _y = True   # valid

 9x = False  # starts with numeral
 => SyntaxError: invalid syntax

 $y = False #  starts with symbol
 => SyntaxError: invalid syntax
```

2. El resto de su nombre de variable puede consistir en letras, números y guiones bajos.

```
has_0_in_it = "Still Valid"
```

3. Los nombres son mayúsculas y minúsculas.

```
x = 9
y = X*5
=>NameError: name 'X' is not defined
```

Aunque no es necesario especificar un tipo de datos al declarar una variable en Python, mientras se asigna el área necesaria en la memoria para la variable, el intérprete de Python selecciona automáticamente el tipo incorporado más adecuado para ella:

```
a = 2
print(type(a))
# Output: <type 'int'>

b = 9223372036854775807
print(type(b))
# Output: <type 'int'>

pi = 3.14
print(type(pi))
# Output: <type 'float'>

c = 'A'
print(type(c))
# Output: <type 'str'>

name = 'John Doe'
print(type(name))
# Output: <type 'str'>

q = True
print(type(q))
# Output: <type 'bool'>

x = None
print(type(x))
# Output: <type 'NoneType'>
```

Ahora que conoce los conceptos básicos de la asignación, dejemos de lado esta sutileza acerca de la asignación en python.

Cuando usa `=` para realizar una operación de asignación, lo que está a la izquierda de `=` es un ***nombre*** para el ***objeto*** a la derecha. Finalmente, lo que hace `=` es asignar la ***referencia*** del objeto a la derecha al ***nombre*** a la izquierda.

Es decir:

```
a_name = an_object  # "a_name" is now a name for the reference to the object "an_object"
```

Entonces, de los muchos ejemplos de asignación anteriores, si seleccionamos `pi = 3.14` , `pi` es **un** nombre (no **el** nombre, ya que un objeto puede tener varios nombres) para el objeto `3.14` . Si no entiende algo a continuación, vuelva a este punto y lea esto nuevamente. Además, puedes echarle un vistazo a esto para una mejor comprensión.

Puede asignar múltiples valores a múltiples variables en una línea. Tenga en cuenta que debe haber el mismo número de argumentos en los lados derecho e izquierdo del operador `=` :

```
a, b, c = 1, 2, 3
print(a, b, c)
# Output: 1 2 3

a, b, c = 1, 2
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b, c = 1, 2
=> ValueError: need more than 2 values to unpack

a, b = 1, 2, 3
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b = 1, 2, 3
=> ValueError: too many values to unpack
```

El error en el último ejemplo puede obviarse asignando valores restantes a un número igual de variables arbitrarias. Esta variable ficticia puede tener cualquier nombre, pero es convencional usar el subrayado (_) para asignar valores no deseados:

```
a, b, _ = 1, 2, 3
print(a, b)
# Output: 1, 2
```

Tenga en cuenta que el número de _ y el número de valores restantes deben ser iguales. De lo contrario, se lanzan 'demasiados valores para descomprimir el error' como se indica arriba:

```
a, b, _ = 1,2,3,4
=>Traceback (most recent call last):
=>File "name.py", line N, in <module>
=>a, b, _ = 1,2,3,4
=>ValueError: too many values to unpack (expected 3)
```

También puede asignar un solo valor a varias variables simultáneamente.

```
a = b = c = 1
print(a, b, c)
# Output: 1 1 1
```

Cuando se utiliza dicha asignación en cascada, es importante tener en cuenta que las tres variables `a` , `b` y `c` se refieren al *mismo objeto* en la memoria, un `int` objeto con el valor de 1. En otras palabras, `a` , `b` y `c` son tres nombres diferentes dado al mismo objeto int. Asignar un objeto diferente a uno de ellos después no cambia los otros, como se esperaba:

```
a = b = c = 1    # all three names a, b and c refer to same int object with value 1
print(a, b, c)
# Output: 1 1 1
b = 2            # b now refers to another int object, one with a value of 2
print(a, b, c)
# Output: 1 2 1  # so output is as expected.
```

Lo anterior también es válido para los tipos mutables (como `list` , `dict` , etc.), al igual que para los tipos inmutables (como `int` , `string` , `tuple` , etc.):

```
x = y = [7, 8, 9]   # x and y refer to the same list object just created, [7, 8, 9]
x = [13, 8, 9]      # x now refers to a different list object just created, [13, 8, 9]
print(y)            # y still refers to the list it was first assigned
# Output: [7, 8, 9]
```

Hasta ahora tan bueno. Las cosas son un poco diferentes cuando se trata de *modificar* el objeto (en contraste con *asignar* el nombre a un objeto diferente, como hicimos anteriormente) cuando la asignación en cascada se usa para tipos mutables. Echa un vistazo a continuación, y lo verás de primera mano:

```
x = y = [7, 8, 9]     # x and y are two different names for the same list object just created,
[7, 8, 9]
x[0] = 13             # we are updating the value of the list [7, 8, 9] through one of its
names, x in this case
print(y)              # printing the value of the list using its other name
# Output: [13, 8, 9]  # hence, naturally the change is reflected
```

Las listas anidadas también son válidas en python. Esto significa que una lista puede contener otra lista como un elemento.

```
x = [1, 2, [3, 4, 5], 6, 7] # this is nested list
print x[2]
# Output: [3, 4, 5]
print x[2][1]
# Output: 4
```

Por último, las variables en Python no tienen que ser del mismo tipo de las que se definieron por primera vez; simplemente puede usar `=` para asignar un nuevo valor a una variable, incluso si ese valor es de un tipo diferente.

```
a = 2
print(a)
# Output: 2

a = "New value"
print(a)
# Output: New value
```

Si esto te molesta, piensa en el hecho de que lo que está a la izquierda de `=` es solo un nombre para un objeto. En primer lugar se llama a la `int` objeto con valor de 2 `a` , a continuación, cambia de opinión y decide dar el nombre `a` a una `string` objeto, que tiene un valor 'Valor nuevo'. Simple, ¿verdad?

Entrada del usuario

Entrada interactiva

Para obtener información del usuario, use la función de `input` (**nota** : en Python 2.x, la función se llama `raw_input` en `raw_input` lugar, aunque Python 2.x tiene su propia versión de `input` que es completamente diferente):

Python 2.x 2.3

```
name = raw_input("What is your name? ")
# Out: What is your name? _
```

> **Observación de seguridad** No use `input()` en Python2: el texto ingresado se evaluará como si fuera una expresión de Python (equivalente a `eval(input())` en Python3), que podría convertirse fácilmente en una vulnerabilidad. Consulte este artículo para obtener más información sobre los riesgos de usar esta función.

Python 3.x 3.0

```
name = input("What is your name? ")
# Out: What is your name? _
```

El resto de este ejemplo utilizará la sintaxis de Python 3.

La función toma un argumento de cadena, que lo muestra como una solicitud y devuelve una cadena. El código anterior proporciona un aviso, esperando que el usuario ingrese.

```
name = input("What is your name? ")
# Out: What is your name?
```

Si el usuario escribe "Bob" y pulsa enter, el `name` la variable se asignará a la cadena `"Bob"` :

```
name = input("What is your name? ")
# Out: What is your name? Bob
print(name)
# Out: Bob
```

Tenga en cuenta que la `input` siempre es de tipo `str` , lo que es importante si desea que el usuario ingrese números. Por lo tanto, necesita convertir el `str` antes de intentar usarlo como un número:

```
x = input("Write a number:")
# Out: Write a number: 10
x / 2
# Out: TypeError: unsupported operand type(s) for /: 'str' and 'int'
float(x) / 2
# Out: 5.0
```

NB: se recomienda utilizar los bloques de `try` / `except` para detectar excepciones cuando se trata de entradas de usuario . Por ejemplo, si su código quiere convertir un `raw_input` en un `int` , y lo que el usuario escribe es inasible, genera un `ValueError` .

IDLE - GUI de Python

IDLE es un entorno integrado de desarrollo y aprendizaje de Python y es una alternativa a la línea de comandos. Como su nombre puede implicar, IDLE es muy útil para desarrollar un nuevo código o aprender Python. En Windows, esto viene con el intérprete de Python, pero en otros sistemas operativos puede que necesite instalarlo a través de su administrador de paquetes.

Los principales propósitos de IDLE son:

- Editor de texto de múltiples ventanas con resaltado de sintaxis, autocompletado y sangría inteligente
- Python shell con resaltado de sintaxis
- Depurador integrado con pasos, puntos de interrupción persistentes y visibilidad de la pila de llamadas
- Sangría automática (útil para los principiantes que aprenden sobre la sangría de Python)
- Guardando el programa Python como archivos .py, ejecútelos y edítelos más tarde en cualquiera de ellos usando IDLE.

En IDLE, presione `F5` o `run Python Shell` para iniciar un intérprete. Usar IDLE puede ser una mejor experiencia de aprendizaje para los nuevos usuarios porque el código se interpreta a medida que el usuario escribe.

Tenga en cuenta que hay muchas alternativas, vea, por ejemplo, esta discusión o esta lista .

Solución de problemas

- **Windows**

  Si estás en Windows, el comando predeterminado es `python` . Si recibe un error `"'python' is not recognized"` Python `"'python' is not recognized"` , la causa más probable es que la ubicación de Python no se encuentre en la `PATH` entorno `PATH` su sistema. Se puede acceder a esto haciendo clic con el botón derecho en 'Mi PC' y seleccionando 'Propiedades' o navegando a 'Sistema' a través de 'Panel de control'. Haga clic en 'Configuración avanzada del sistema' y luego en 'Variables de entorno ...'. Edite la variable `PATH` para incluir el

directorio de su instalación de Python, así como la carpeta Script (generalmente `C:\Python27;C:\Python27\Scripts`). Esto requiere privilegios administrativos y puede requerir un reinicio.

Cuando se usan varias versiones de Python en la misma máquina, una posible solución es cambiar el nombre de uno de los archivos `python.exe` . Por ejemplo, nombrar una versión `python27.exe` haría que `python27` convierta en el comando de Python para esa versión.

También puede usar el Lanzador de Python para Windows, que está disponible a través del instalador y viene por defecto. Le permite seleccionar la versión de Python para ejecutar usando `py -[xy]` lugar de `python[xy]` . Puede usar la última versión de Python 2 ejecutando scripts con `py -2` y la última versión de Python 3 ejecutando scripts con `py -3` .

- **Debian / Ubuntu / MacOS**

  Esta sección asume que la ubicación del ejecutable de `python` se ha agregado a la `PATH` entorno `PATH` .

  Si estás en Debian / Ubuntu / MacOS, abre el terminal y escribe `python` para Python 2.xo `python3` para Python 3.x.

  Escriba `which python` para ver qué intérprete de Python se utilizará.

- **Arco de linux**

  El Python predeterminado en Arch Linux (y sus descendientes) es Python 3, así que usa `python` o `python3` para Python `python2` para Python 2.x.

- **Otros sistemas**

  Python 3 a veces está vinculado a `python` lugar de `python3` . Para usar Python 2 en estos sistemas donde está instalado, puede usar `python2` .

Tipos de datos

Tipos incorporados

Booleanos

`bool` : Un valor booleano de `True` o `False` . Las operaciones lógicas como `and` , `or` , `not` se pueden realizar en valores booleanos.

```
x or y    # if x is False then y otherwise x
x and y   # if x is False then x otherwise y
not x     # if x is True then False, otherwise True
```

En Python 2.xy en Python 3.x, un booleano es también un `int` . El tipo `bool` es una subclase del tipo `int` y `True` y `False` son sus únicas instancias:

```
issubclass(bool, int) # True

isinstance(True, bool) # True
isinstance(False, bool) # True
```

Si se usan valores booleanos en operaciones aritméticas, sus valores enteros (`1` y `0` para `True` y `False`) se usarán para devolver un resultado entero:

```
True + False == 1 # 1 + 0 == 1
True * True  == 1 # 1 * 1 == 1
```

Números

- `int` : número entero

  ```
  a = 2
  b = 100
  c = 123456789
  d = 38563846326424324
  ```

  Los enteros en Python son de tamaños arbitrarios.

  Nota: en versiones anteriores de Python, había un tipo `long` disponible y esto era distinto de `int` . Los dos se han unificado.

- `float` : número de punto flotante; La precisión depende de la implementación y la arquitectura del sistema, para CPython el tipo de datos `float` corresponde a un C doble.

  ```
  a = 2.0
  b = 100.e0
  c = 123456789.e1
  ```

- `complex` : números complejos

  ```
  a = 2 + 1j
  b = 100 + 10j
  ```

Los operadores `<` , `<=` , `>` y `>=` generarán una excepción `TypeError` cuando cualquier operando sea un número complejo.

Instrumentos de cuerda

Python 3.x 3.0

- `str` : una **cadena de Unicode** . El tipo de `'hello'`
- `bytes` : una **cadena de bytes** . El tipo de `b'hello'`

Python 2.x 2.7

- `str` : una **cadena de bytes** . El tipo de `'hello'`
- `bytes` : sinónimo de `str`
- `unicode` : una **cadena de Unicode** . El tipo de `u'hello'`

Secuencias y colecciones.

Python diferencia entre secuencias ordenadas y colecciones no ordenadas (como `set` y `dict`).

- Las cadenas (`str` , `bytes` , `unicode`) son secuencias.
- `reversed` : un orden invertido de `str` con función `reversed`

```
a = reversed('hello')
```

- `tuple` : una colección ordenada de `n` valores de cualquier tipo (`n >= 0`).

```
a = (1, 2, 3)
b = ('a', 1, 'python', (1, 2))
b[2] = 'something else' # returns a TypeError
```

Soporta indexación; inmutable; hashable si todos sus miembros son hashable

- `list` : una colección ordenada de `n` valores (`n >= 0`)

```
a = [1, 2, 3]
b = ['a', 1, 'python', (1, 2), [1, 2]]
b[2] = 'something else' # allowed
```

No hashable; mudable.

- `set` : Una colección desordenada de valores únicos. Los artículos deben ser hashable .

```
a = {1, 2, 'a'}
```

- `dict` : una colección desordenada de pares clave-valor únicos; Las claves deben ser hashable .

```
a = {1: 'one',
     2: 'two'}

b = {'a': [1, 2, 3],
     'b': 'a string'}
```

Un objeto es hashable si tiene un valor hash que nunca cambia durante su vida útil (necesita un `__hash__()`) y puede compararse con otros objetos (necesita un `__eq__()`). Los objetos hashable que comparan la igualdad deben tener el mismo valor hash.

Constantes incorporadas

Junto con los tipos de datos incorporados, hay una pequeña cantidad de constantes incorporadas en el espacio de nombres integrado:

- `True` : El valor verdadero del tipo incorporado `bool`
- `False` : el valor falso del tipo incorporado `bool`
- `None` : un objeto singleton utilizado para indicar que un valor está ausente.
- `Ellipsis` o `...` : se utiliza en Python3 + en cualquier lugar y uso limitado en Python2.7 + como parte de la notación de matriz. `numpy` paquetes `numpy` y relacionados usan esto como una referencia 'incluir todo' en los arreglos.
- `NotImplemented` : un singleton utilizado para indicar a Python que un método especial no es compatible con los argumentos específicos, y Python probará alternativas si están disponibles.

```
a = None # No value will be assigned. Any valid datatype can be assigned later
```

Python 3.x 3.0

`None` no tiene ningún orden natural. El uso de operadores de comparación de pedidos (`<` , `<=` , `>=` , `>`) ya no es compatible y generará un `TypeError` .

Python 2.x 2.7

`None` siempre es menor que cualquier número (`None < -32` evalúa como `True`).

Probando el tipo de variables

En Python, podemos verificar el tipo de datos de un objeto usando el `type` función incorporada.

```
a = '123'
print(type(a))
# Out: <class 'str'>
b = 123
print(type(b))
# Out: <class 'int'>
```

En declaraciones condicionales es posible probar el tipo de datos con `isinstance` . Sin embargo, generalmente no se recomienda confiar en el tipo de variable.

```
i = 7
if isinstance(i, int):
    i += 1
elif isinstance(i, str):
    i = int(i)
    i += 1
```

Para obtener información sobre las diferencias entre `type()` y `isinstance()` lea: Diferencias entre isinstance y type en Python

Para probar si algo es de `NoneType` :

```
x = None
if x is None:
    print('Not a surprise, I just defined x as None.')
```

Convertir entre tipos de datos

Puede realizar la conversión explícita de tipos de datos.

Por ejemplo, '123' es de tipo `str` y se puede convertir en entero usando la función `int` .

```
a = '123'
b = int(a)
```

La conversión de una cadena flotante como '123.456' se puede hacer usando la función `float` .

```
a = '123.456'
b = float(a)
c = int(a)    # ValueError: invalid literal for int() with base 10: '123.456'
d = int(b)    # 123
```

También puedes convertir secuencias o tipos de colección.

```
a = 'hello'
list(a)  # ['h', 'e', 'l', 'l', 'o']
set(a)   # {'o', 'e', 'l', 'h'}
tuple(a) # ('h', 'e', 'l', 'l', 'o')
```

Tipo de cadena explícita en la definición de literales

Con las etiquetas de una letra justo delante de las comillas, puede indicar qué tipo de cadena desea definir.

- `b'foo bar'` : `bytes` resultados en Python 3, `str` en Python 2
- `u'foo bar'` : results `str` en Python 3, `unicode` en Python 2
- `'foo bar'` : resultados `str`
- `r'foo bar'` : resultados llamados cadenas en bruto, donde no es necesario escapar caracteres especiales, todo se toma literalmente a medida que se escribe

```
normal  = 'foo\nbar'   # foo
                       # bar
escaped = 'foo\\nbar'  # foo\nbar
raw     = r'foo\nbar'  # foo\nbar
```

Tipos de datos mutables e inmutables

Un objeto se llama *mutable* si se puede cambiar. Por ejemplo, cuando pasa una lista a alguna función, la lista se puede cambiar:

```
def f(m):
    m.append(3)  # adds a number to the list. This is a mutation.

x = [1, 2]
f(x)
x == [1, 2]  # False now, since an item was added to the list
```

Un objeto se llama *inmutable* si no se puede cambiar de ninguna manera. Por ejemplo, los enteros son inmutables, ya que no hay forma de cambiarlos:

```
def bar():
    x = (1, 2)
    g(x)
    x == (1, 2)  # Will always be True, since no function can change the object (1, 2)
```

Tenga en cuenta que las **variables en** sí mismas son mutables, por lo que podemos reasignar la *variable* `x` , pero esto no cambia el objeto al que `x` había apuntado anteriormente. Solo hizo que `x` apunte a un nuevo objeto.

Los tipos de datos cuyas instancias son mutables se denominan *tipos de datos mutables* , y de manera similar para los objetos y tipos de datos inmutables.

Ejemplos de tipos de datos inmutables:

- `int` , `long` , `float` , `complex`
- `str`
- `bytes`
- `tuple`
- `frozenset`

Ejemplos de tipos de datos mutables:

- `bytearray`
- `list`
- `set`
- `dict`

Construido en módulos y funciones

Un módulo es un archivo que contiene definiciones y declaraciones de Python. La función es un fragmento de código que ejecuta alguna lógica.

```
>>> pow(2,3)    #8
```

Para verificar la función incorporada en python podemos usar `dir()`. Si se llama sin un

argumento, devuelve los nombres en el alcance actual. De lo contrario, devuelve una lista alfabética de nombres que comprenden (algunos de) el atributo del objeto dado y de los atributos a los que se puede acceder.

```
>>> dir(__builtins__)
[
    'ArithmeticError',
    'AssertionError',
    'AttributeError',
    'BaseException',
    'BufferError',
    'BytesWarning',
    'DeprecationWarning',
    'EOFError',
    'Ellipsis',
    'EnvironmentError',
    'Exception',
    'False',
    'FloatingPointError',
    'FutureWarning',
    'GeneratorExit',
    'IOError',
    'ImportError',
    'ImportWarning',
    'IndentationError',
    'IndexError',
    'KeyError',
    'KeyboardInterrupt',
    'LookupError',
    'MemoryError',
    'NameError',
    'None',
    'NotImplemented',
    'NotImplementedError',
    'OSError',
    'OverflowError',
    'PendingDeprecationWarning',
    'ReferenceError',
    'RuntimeError',
    'RuntimeWarning',
    'StandardError',
    'StopIteration',
    'SyntaxError',
    'SyntaxWarning',
    'SystemError',
    'SystemExit',
    'TabError',
    'True',
    'TypeError',
    'UnboundLocalError',
    'UnicodeDecodeError',
    'UnicodeEncodeError',
    'UnicodeError',
    'UnicodeTranslateError',
    'UnicodeWarning',
    'UserWarning',
    'ValueError',
    'Warning',
    'ZeroDivisionError',
    '__debug__',
```

```
    '__doc__',
    '__import__',
    '__name__',
    '__package__',
    'abs',
    'all',
    'any',
    'apply',
    'basestring',
    'bin',
    'bool',
    'buffer',
    'bytearray',
    'bytes',
    'callable',
    'chr',
    'classmethod',
    'cmp',
    'coerce',
    'compile',
    'complex',
    'copyright',
    'credits',
    'delattr',
    'dict',
    'dir',
    'divmod',
    'enumerate',
    'eval',
    'execfile',
    'exit',
    'file',
    'filter',
    'float',
    'format',
    'frozenset',
    'getattr',
    'globals',
    'hasattr',
    'hash',
    'help',
    'hex',
    'id',
    'input',
    'int',
    'intern',
    'isinstance',
    'issubclass',
    'iter',
    'len',
    'license',
    'list',
    'locals',
    'long',
    'map',
    'max',
    'memoryview',
    'min',
    'next',
    'object',
    'oct',
```

```
    'open',
    'ord',
    'pow',
    'print',
    'property',
    'quit',
    'range',
    'raw_input',
    'reduce',
    'reload',
    'repr',
    'reversed',
    'round',
    'set',
    'setattr',
    'slice',
    'sorted',
    'staticmethod',
    'str',
    'sum',
    'super',
    'tuple',
    'type',
    'unichr',
    'unicode',
    'vars',
    'xrange',
    'zip'
]
```

Para conocer la funcionalidad de cualquier función, podemos utilizar la `help` incorporada de la función.

```
>>> help(max)
Help on built-in function max in module __builtin__:
max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
```

Los módulos incorporados contienen funcionalidades adicionales. Por ejemplo, para obtener la raíz cuadrada de un número, necesitamos incluir `math` módulo `math` .

```
>>> import math
>>> math.sqrt(16) # 4.0
```

Para conocer todas las funciones de un módulo, podemos asignar la lista de funciones a una variable y luego imprimir la variable.

```
>>> import math
>>> dir(math)

   ['__doc__', '__name__', '__package__', 'acos', 'acosh',
   'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
   'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
```

```
   'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
   'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
   'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
   'tan', 'tanh', 'trunc']
```

Parece que `__doc__` es útil para proporcionar alguna documentación en, digamos, funciones

```
>>> math.__doc__
'This module is always available.  It provides access to the\nmathematical
 functions defined by the C standard.'
```

Además de las funciones, la documentación también se puede proporcionar en módulos. Entonces, si tienes un archivo llamado `helloWorld.py` como este:

```
"""This is the module docstring."""

def sayHello():
    """This is the function docstring."""
    return 'Hello World'
```

Puedes acceder a sus documentos como este:

```
>>> import helloWorld
>>> helloWorld.__doc__
'This is the module docstring.'
>>> helloWorld.sayHello.__doc__
'This is the function docstring.'
```

- Para cualquier tipo definido por el usuario, sus atributos, los atributos de su clase y recursivamente los atributos de las clases base de su clase se pueden recuperar usando dir ()

```
>>> class MyClassObject(object):
...     pass
...
>>> dir(MyClassObject)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Cualquier tipo de datos se puede convertir simplemente a cadena usando una función incorporada llamada `str` . Esta función se llama de forma predeterminada cuando se pasa un tipo de datos para `print`

```
>>> str(123)    # "123"
```

Sangría de bloque

Python utiliza la sangría para definir el control y las construcciones de bucle. Esto contribuye a la legibilidad de Python, sin embargo, requiere que el programador preste mucha atención al uso del espacio en blanco. Por lo tanto, la mala calibración del editor podría resultar en un código que se

comporta de manera inesperada.

Python usa el símbolo de dos puntos (`:`) y sangría para mostrar dónde bloques de código empiezan y terminan (Si viene de otro idioma, no confunda esto con alguna manera estar relacionado con el operador ternario). Es decir, los bloques en Python, tales como funciones, bucles, cláusulas `if` y otras construcciones, no tienen identificadores finales. Todos los bloques comienzan con dos puntos y luego contienen las líneas sangradas debajo de él.

Por ejemplo:

```
def my_function():    # This is a function definition. Note the colon (:)
    a = 2             # This line belongs to the function because it's indented
    return a          # This line also belongs to the same function
print(my_function())  # This line is OUTSIDE the function block
```

o

```
if a > b:             # If block starts here
    print(a)          # This is part of the if block
else:                 # else must be at the same level as if
    print(b)          # This line is part of the else block
```

Los bloques que contienen exactamente una instrucción de una sola línea se pueden colocar en la misma línea, aunque esta forma generalmente no se considera un buen estilo:

```
if a > b: print(a)
else: print(b)
```

Intentar hacer esto con más de una sola declaración *no* funcionará:

```
if x > y: y = x
    print(y) # IndentationError: unexpected indent

if x > y: while y != z: y -= 1  # SyntaxError: invalid syntax
```

Un bloque vacío causa un `IndentationError` . Use `pass` (un comando que no hace nada) cuando tiene un bloque sin contenido:

```
def will_be_implemented_later():
    pass
```

Espacios vs. Pestañas

En resumen: **siempre** usa 4 espacios para la sangría.

El uso exclusivo de pestañas es posible, pero PEP 8 , la guía de estilo para el código Python, indica que se prefieren los espacios.

Python 3.x 3.0

Python 3 no permite mezclar el uso de tabulaciones y espacios para la sangría. En tal caso, se genera un error en tiempo de compilación: `Inconsistent use of tabs and spaces in indentation` y el programa no se ejecutará.

Python 2.x 2.7

Python 2 permite mezclar tabulaciones y espacios en sangría; Esto es fuertemente desalentado. El carácter de la pestaña completa la sangría anterior para ser un múltiplo de **8** espacios . Como es común que los editores estén configurados para mostrar pestañas como múltiplos de **4** espacios, esto puede causar errores sutiles.

Citando el PEP 8 :

> Cuando se invoca al intérprete de línea de comandos de Python 2 con la opción `-t` , emite advertencias sobre el código que mezcla de forma ilegal las pestañas y los espacios. Al usar `-tt` estas advertencias se convierten en errores. Estas opciones son muy recomendables!

Muchos editores tienen configuración de "pestañas a espacios". Al configurar el editor, uno debe diferenciar entre el *carácter de* la pestaña ('\ t') y la tecla `Tab` .

- El *carácter de la* pestaña debe configurarse para mostrar 8 espacios, para que coincida con la semántica del idioma, al menos en los casos en que es posible una sangría (accidental). Los editores también pueden convertir automáticamente el carácter de tabulación a espacios.
- Sin embargo, puede ser útil configurar el editor para que al presionar la tecla `Tab` , se inserten 4 espacios, en lugar de insertar un carácter de pestaña.

El código fuente de Python escrito con una combinación de tabulaciones y espacios, o con un número no estándar de espacios de sangría se puede hacer compatible con pep8 usando autopep8 . (Una alternativa menos poderosa viene con la mayoría de las instalaciones de Python: reindent.py)

Tipos de colección

Hay varios tipos de colecciones en Python. Mientras que los tipos como `int` y `str` tienen un solo valor, los tipos de colección tienen múltiples valores.

Liza

El tipo de `list` es probablemente el tipo de colección más utilizado en Python. A pesar de su nombre, una lista es más como una matriz en otros idiomas, principalmente JavaScript. En Python, una lista es simplemente una colección ordenada de valores válidos de Python. Se puede crear una lista encerrando valores, separados por comas, entre corchetes:

```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

Una lista puede estar vacía:

```
empty_list = []
```

Los elementos de una lista no están restringidos a un solo tipo de datos, lo que tiene sentido dado que Python es un lenguaje dinámico:

```
mixed_list = [1, 'abc', True, 2.34, None]
```

Una lista puede contener otra lista como su elemento:

```
nested_list = [['a', 'b', 'c'], [1, 2, 3]]
```

Se puede acceder a los elementos de una lista a través de un *índice* o representación numérica de su posición. Las listas en Python tienen *índice cero,* lo que significa que el primer elemento de la lista está en el índice 0, el segundo elemento está en el índice 1 y así sucesivamente:

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
```

Los índices también pueden ser negativos, lo que significa contar desde el final de la lista (`-1` es el índice del último elemento). Entonces, usando la lista del ejemplo anterior:

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

Las listas son mutables, por lo que puede cambiar los valores en una lista:

```
names[0] = 'Ann'
print(names)
# Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

Además, es posible agregar y / o eliminar elementos de una lista:

Agregar objeto al final de la lista con `L.append(object)` , devuelve `None` .

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# Outputs ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Agregue un nuevo elemento a la lista en un índice específico. `L.insert(index, object)`

```
names.insert(1, "Nikki")
print(names)
# Outputs ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Elimine la primera aparición de un valor con `L.remove(value)` , devuelve `None`

```
names.remove("Bob")
```

```
print(names) # Outputs ['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

Obtenga el índice en la lista del primer elemento cuyo valor es x. Se mostrará un error si no hay tal elemento.

```
name.index("Alice")
0
```

Cuenta la longitud de la lista

```
len(names)
6
```

cuenta la ocurrencia de cualquier artículo en la lista

```
a = [1, 1, 1, 2, 3, 4]
a.count(1)
3
```

Revertir la lista

```
a.reverse()
[4, 3, 2, 1, 1, 1]
# or
a[::-1]
[4, 3, 2, 1, 1, 1]
```

Elimine y devuelva el elemento en el índice (predeterminado al último elemento) con `L.pop([index])` , devuelve el elemento

```
names.pop() # Outputs 'Sia'
```

Puede iterar sobre los elementos de la lista como a continuación:

```
for element in my_list:
    print (element)
```

Tuplas

Una `tuple` es similar a una lista, excepto que es de longitud fija e inmutable. Por lo tanto, los valores de la tupla no se pueden cambiar ni los valores se pueden agregar o quitar de la tupla. Las tuplas se usan comúnmente para pequeñas colecciones de valores que no será necesario cambiar, como una dirección IP y un puerto. Las tuplas se representan con paréntesis en lugar de corchetes:

```
ip_address = ('10.20.30.40', 8080)
```

Las mismas reglas de indexación para las listas también se aplican a las tuplas. Las tuplas también se pueden anidar y los valores pueden ser válidos para cualquier Python válido.

Una tupla con un solo miembro debe definirse (tenga en cuenta la coma) de esta manera:

```
one_member_tuple = ('Only member',)
```

o

```
one_member_tuple = 'Only member',   # No brackets
```

o simplemente usando la sintaxis de la `tuple`

```
one_member_tuple = tuple(['Only member'])
```

Los diccionarios

Un `dictionary` en Python es una colección de pares clave-valor. El diccionario está rodeado de llaves. Cada par está separado por una coma y la clave y el valor están separados por dos puntos. Aquí hay un ejemplo:

```
state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

Para obtener un valor, consúltelo por su clave:

```
ca_capital = state_capitals['California']
```

También puede obtener todas las claves en un diccionario y luego repetirlas:

```
for k in state_capitals.keys():
    print('{} is the capital of {}'.format(state_capitals[k], k))
```

Los diccionarios se parecen mucho a la sintaxis JSON. El módulo `json` nativo en la biblioteca estándar de Python se puede usar para convertir entre JSON y diccionarios.

conjunto

Un `set` es una colección de elementos sin repeticiones y sin orden de inserción pero ordenado. Se usan en situaciones en las que solo es importante que algunas cosas se agrupen y no en qué orden se incluyeron. Para grupos grandes de datos, es mucho más rápido verificar si un elemento está o no en un `set` que hacer lo mismo para una `list` .

Definir un `set` es muy similar a definir un `dictionary` :

```
first_names = {'Adam', 'Beth', 'Charlie'}
```

O puedes construir un `set` usando una `list` existente:

```
my_list = [1,2,3]
my_set = set(my_list)
```

Verifique la membresía del `set` usando `in` :

```
if name in first_names:
    print(name)
```

Puede iterar sobre un `set` exactamente como una lista, pero recuerde: los valores estarán en un orden arbitrario, definido por la implementación.

sentencia por defecto

Un `defaultdict` es un diccionario con un valor por defecto para las llaves, por lo que las claves para el que ha sido definida explícitamente ningún valor se puede acceder sin errores. `defaultdict` es especialmente útil cuando los valores del diccionario son colecciones (listas, dicts, etc.) en el sentido de que no es necesario inicializar cada vez que se usa una nueva clave.

Un `defaultdict` nunca generará un KeyError. Cualquier clave que no exista obtiene el valor predeterminado devuelto.

Por ejemplo, considere el siguiente diccionario

```
>>> state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

Si intentamos acceder a una clave que no existe, Python nos devuelve un error de la siguiente manera

```
>>> state_capitals['Alabama']
Traceback (most recent call last):

  File "<ipython-input-61-236329695e6f>", line 1, in <module>
    state_capitals['Alabama']

KeyError: 'Alabama'
```

Probemos con un `defaultdict` . Se puede encontrar en el módulo de colecciones.

```
>>> from collections import defaultdict
>>> state_capitals = defaultdict(lambda: 'Boston')
```

Lo que hicimos aquí es establecer un valor predeterminado (**Boston**) en caso de que la clave de asignación no exista. Ahora rellena el dict como antes:

```
>>> state_capitals['Arkansas'] = 'Little Rock'
>>> state_capitals['California'] = 'Sacramento'
```

```
>>> state_capitals['Colorado'] = 'Denver'
>>> state_capitals['Georgia'] = 'Atlanta'
```

Si intentamos acceder al dict con una clave que no existe, Python nos devolverá el valor predeterminado, es decir, Boston

```
>>> state_capitals['Alabama']
'Boston'
```

y devuelve los valores creados para la clave existente al igual que un `dictionary` normal

```
>>> state_capitals['Arkansas']
'Little Rock'
```

Utilidad de ayuda

Python tiene varias funciones integradas en el intérprete. Si desea obtener información sobre palabras clave, funciones incorporadas, módulos o temas, abra una consola de Python e ingrese:

```
>>> help()
```

Recibirá información ingresando palabras clave directamente:

```
>>> help(help)
```

o dentro de la utilidad:

```
help> help
```

que mostrará una explicación:

```
Help on _Helper in module _sitebuiltins object:

class _Helper(builtins.object)
 |  Define the builtin 'help'.
 |
 |  This is a wrapper around pydoc.help that provides a helpful message
 |  when 'help' is typed at the Python interactive prompt.
 |
 |  Calling help() at the Python prompt starts an interactive help session.
 |  Calling help(thing) prints help for the python object 'thing'.
 |
 |  Methods defined here:
 |
 |  __call__(self, *args, **kwds)
 |
 |  __repr__(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
```

```
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

También puedes solicitar subclases de módulos:

```
help(pymysql.connections)
```

Puede usar la ayuda para acceder a las cadenas de documentación de los diferentes módulos que ha importado, por ejemplo, intente lo siguiente:

```
>>> help(math)
```

y obtendrás un error

```
>>> import math
>>> help(math)
```

Y ahora obtendrá una lista de los métodos disponibles en el módulo, pero solo DESPUÉS de haberlo importado.

Cerrar el ayudante con `quit`

Creando un modulo

Un módulo es un archivo importable que contiene definiciones y declaraciones.

Se puede crear un módulo creando un archivo `.py` .

```
# hello.py
def say_hello():
    print("Hello!")
```

Las funciones en un módulo se pueden utilizar importando el módulo.

Para los módulos que haya creado, deberán estar en el mismo directorio que el archivo en el que los está importando. (Sin embargo, también puede colocarlos en el directorio lib de Python con los módulos pre-incluidos, pero debe evitarse si es posible).

```
$ python
>>> import hello
>>> hello.say_hello()
=> "Hello!"
```

Los módulos pueden ser importados por otros módulos.

```
# greet.py
import hello
hello.say_hello()
```

Se pueden importar funciones específicas de un módulo.

```
# greet.py
from hello import say_hello
say_hello()
```

Los módulos pueden ser alias.

```
# greet.py
import hello as ai
ai.say_hello()
```

Un módulo puede ser un script ejecutable independiente.

```
# run_hello.py
if __name__ == '__main__':
    from hello import say_hello
    say_hello()
```

¡Ejecutarlo!

```
$ python run_hello.py
=> "Hello!"
```

Si el módulo está dentro de un directorio y necesita ser detectado por python, el directorio debe contener un archivo llamado `__init__.py` .

Función de cadena - str () y repr ()

Hay dos funciones que se pueden usar para obtener una representación legible de un objeto.

`repr(x)` llama a `x.__repr__()` : una representación de `x` . `eval` general, `eval` convertirá el resultado de esta función al objeto original.

`str(x)` llama a `x.__str__()` : una cadena legible por humanos que describe el objeto. Esto puede ocultar algunos detalles técnicos.

---

repr ()

Para muchos tipos, esta función intenta devolver una cadena que produciría un objeto con el mismo valor cuando se pasa a `eval()` . De lo contrario, la representación es una cadena encerrada entre paréntesis angulares que contiene el nombre del tipo del objeto junto con información adicional. Esto a menudo incluye el nombre y la dirección del objeto.

str ()

Para las cadenas, esto devuelve la cadena en sí. La diferencia entre esto y `repr(object)` es que `str(object)` no siempre intenta devolver una cadena que sea aceptable para `eval()` . Más bien, su

objetivo es devolver una cadena imprimible o "legible para humanos". Si no se da ningún argumento, esto devuelve la cadena vacía, `''` .

Ejemplo 1:

```
s = """w'o"w"""
repr(s) # Output: '\'w\\\'o"w\''
str(s)  # Output: 'w\'o"w'
eval(str(s)) == s  # Gives a SyntaxError
eval(repr(s)) == s # Output: True
```

Ejemplo 2:

```
import datetime
today = datetime.datetime.now()
str(today)  # Output: '2016-09-15 06:58:46.915000'
repr(today) # Output: 'datetime.datetime(2016, 9, 15, 6, 58, 46, 915000)'
```

Al escribir una clase, puede anular estos métodos para hacer lo que quiera:

```
class Represent(object):

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "Represent(x={},y=\"{}\")".format(self.x, self.y)

    def __str__(self):
        return "Representing x as {} and y as {}".format(self.x, self.y)
```

Usando la clase anterior podemos ver los resultados:

```
r = Represent(1, "Hopper")
print(r)  # prints __str__
print(r.__repr__)  # prints __repr__: '<bound method Represent.__repr__ of
Represent(x=1,y="Hopper")>'
rep = r.__repr__()  # sets the execution of __repr__ to a new variable
print(rep)  # prints 'Represent(x=1,y="Hopper")'
r2 = eval(rep) # evaluates rep
print(r2)  # prints __str__ from new object
print(r2 == r)  # prints 'False' because they are different objects
```

Instalación de módulos externos utilizando pip

`pip` es tu amigo cuando necesitas instalar cualquier paquete de la gran cantidad de opciones disponibles en el índice del paquete python (PyPI). `pip` ya está instalado si estás usando Python 2> = 2.7.9 o Python 3> = 3.4 descargado desde python.org. Para computadoras que ejecutan Linux u otro * nix con un administrador de paquetes nativo, `pip` debe instalarse manualmente.

En instancias con Python 2 y Python 3 instalados, `pip` menudo se refiere a Python 2 y `pip3` a Python 3. El uso de `pip` solo instalará paquetes para Python 2 y `pip3` solo instalará paquetes para

Python 3.

Encontrar / instalar un paquete

La búsqueda de un paquete es tan simple como escribir

```
$ pip search <query>
# Searches for packages whose name or summary contains <query>
```

Instalar un paquete es tan simple como escribirlo *(en un terminal / símbolo del sistema, no en el intérprete de Python)*

```
$ pip install [package_name]           # latest version of the package

$ pip install [package_name]==x.x.x    # specific version of the package

$ pip install '[package_name]>=x.x.x'  # minimum version of the package
```

donde `xxx` es el número de versión del paquete que desea instalar.

Cuando su servidor está detrás de proxy, puede instalar el paquete usando el siguiente comando:

```
$ pip --proxy http://<server address>:<port> install
```

Actualización de paquetes instalados

Cuando aparecen nuevas versiones de paquetes instalados, no se instalan automáticamente en su sistema. Para obtener una descripción general de cuáles de sus paquetes instalados están desactualizados, ejecute:

```
$ pip list --outdated
```

Para actualizar el uso de un paquete específico

```
$ pip install [package_name] --upgrade
```

Actualizar todos los paquetes obsoletos no es una funcionalidad estándar de `pip` .

Actualización de pip

Puede actualizar su instalación pip existente usando los siguientes comandos

- En Linux o macOS X:

```
$ pip install -U pip
```

Es posible que necesite usar `sudo` con pip en algunos sistemas Linux

- En Windows:

```
py -m pip install -U pip
```

o

```
python -m pip install -U pip
```

Para más información sobre pip, lea aquí .

Instalación de Python 2.7.xy 3.x

Nota : las siguientes instrucciones están escritas para Python 2.7 (a menos que se especifique): las instrucciones para Python 3.x son similares.

VENTANAS

Primero, descargue la última versión de Python 2.7 del sitio web oficial (https://www.python.org/downloads/) . La versión se proporciona como un paquete MSI. Para instalarlo manualmente, simplemente haga doble clic en el archivo.

Por defecto, Python se instala en un directorio:

```
C:\Python27\
```

Advertencia: la instalación no modifica automáticamente la variable de entorno PATH.

Suponiendo que su instalación de Python está en C: \ Python27, agregue esto a su PATH:

```
C:\Python27\;C:\Python27\Scripts\
```

Ahora para comprobar si la instalación de Python es válida, escriba en cmd:

```
python --version
```

Python 2.xy 3.x lado a lado

Para instalar y usar Python 2.xy 3.x lado a lado en una máquina con Windows:

1. Instale Python 2.x usando el instalador MSI.
    - Asegúrese de que Python esté instalado para todos los usuarios.
    - Opcional: agregue Python a `PATH` para hacer que Python 2.x se pueda llamar desde la línea de comandos usando `python` .

2. Instala Python 3.x usando su respectivo instalador.

    - Una vez más, asegúrese de que Python esté instalado para todos los usuarios.
    - Opcional: agregue Python a `PATH` para que Python 3.x se pueda llamar desde la línea de comandos usando `python` . Esto puede anular la configuración de Python 2.x `PATH` , por lo tanto, vuelva a verificar su `PATH` y asegúrese de que esté configurado según sus preferencias.
    - Asegúrate de instalar el `py launcher` para todos los usuarios.

Python 3 instalará el iniciador de Python que se puede usar para lanzar Python 2.xy Python 3.x indistintamente desde la línea de comandos:

```
P:\>py -3
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

C:\>py -2
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Para usar la versión correspondiente de `pip` para una versión específica de Python, use:

```
C:\>py -3 -m pip -V
pip 9.0.1 from C:\Python36\lib\site-packages (python 3.6)

C:\>py -2 -m pip -V
pip 9.0.1 from C:\Python27\lib\site-packages (python 2.7)
```

LINUX

Las últimas versiones de CentOS, Fedora, Redhat Enterprise (RHEL) y Ubuntu vienen con Python 2.7.

Para instalar Python 2.7 en linux manualmente, simplemente haga lo siguiente en la terminal:

```
wget --no-check-certificate https://www.python.org/ftp/python/2.7.X/Python-2.7.X.tgz
tar -xzf Python-2.7.X.tgz
cd Python-2.7.X
./configure
make
sudo make install
```

También agregue la ruta del nuevo python en la variable de entorno PATH. Si el nuevo python está en `/root/python-2.7.X` , ejecute `export PATH = $PATH:/root/python-2.7.X`

Ahora para comprobar si la instalación de Python es válida, escriba en el terminal:

```
python --version
```

Ubuntu (desde la fuente)

Si necesita Python 3.6, puede instalarlo desde la fuente como se muestra a continuación (Ubuntu 16.10 y 17.04 tienen la versión 3.6 en el repositorio universal). Deben seguirse los siguientes pasos para Ubuntu 16.04 y versiones inferiores:

```
sudo apt install build-essential checkinstall
sudo apt install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev
libgdbm-dev libc6-dev libbz2-dev
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tar.xz
tar xvf Python-3.6.1.tar.xz
cd Python-3.6.1/
./configure --enable-optimizations
sudo make altinstall
```

Mac OS

Mientras hablamos, macOS viene instalado con Python 2.7.10, pero esta versión está desactualizada y ligeramente modificada de Python regular.

> La versión de Python que se incluye con OS X es excelente para el aprendizaje, pero no es buena para el desarrollo. La versión enviada con OS X puede estar desactualizada de la versión oficial actual de Python, que se considera la versión de producción estable. (fuente)

Instalar Homebrew :

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Instala Python 2.7:

```
brew install python
```

Para Python 3.x, use el comando `brew install python3` en `brew install python3` lugar.

Lea Empezando con Python Language en línea:
https://riptutorial.com/es/python/topic/193/empezando-con-python-language

Capítulo 2: * args y ** kwargs

Observaciones

Hay algunas cosas a tener en cuenta:

1. Los nombres `args` y `kwargs` se usan por convención, no forman parte de la especificación del lenguaje. Por lo tanto, estos son equivalentes:

```
def func(*args, **kwargs):
    print(args)
    print(kwargs)
```

```
def func(*a, **b):
    print(a)
    print(b)
```

2. No puede tener más de un `args` o más de un parámetro de `kwargs` (sin embargo, no son necesarios)

```
def func(*args1, *args2):
#   File "<stdin>", line 1
#     def test(*args1, *args2):
#                      ^
# SyntaxError: invalid syntax
```

```
def test(**kwargs1, **kwargs2):
#   File "<stdin>", line 1
#     def test(**kwargs1, **kwargs2):
#                       ^
# SyntaxError: invalid syntax
```

3. Si algún argumento posicional sigue a `*args` , son argumentos de palabra clave que solo se pueden pasar por nombre. Se puede usar una sola estrella en lugar de `*args` para forzar que los valores sean argumentos de palabras clave sin proporcionar una lista de parámetros variadic. Las listas de parámetros de palabras clave solo están disponibles en Python 3.

```
def func(a, b, *args, x, y):
    print(a, b, args, x, y)

func(1, 2, 3, 4, x=5, y=6)
#>>> 1, 2, (3, 4), 5, 6
```

```
def func(a, b, *, x, y):
```

```
    print(a, b, x, y)

func(1, 2, x=5, y=6)
#>>> 1, 2, 5, 6
```

4. `**kwargs` deben estar en último lugar en la lista de parámetros.

```
def test(**kwargs, *args):
#   File "<stdin>", line 1
#     def test(**kwargs, *args):
#                        ^
# SyntaxError: invalid syntax
```

Examples

Usando * args al escribir funciones

Puede usar la estrella * al escribir una función para recopilar todos los argumentos posicionales (es decir, sin nombre) en una tupla:

```
def print_args(farg, *args):
   print("formal arg: %s" % farg)
   for arg in args:
       print("another positional arg: %s" % arg)
```

Método de llamada:

```
print_args(1, "two", 3)
```

En esa llamada, farg se asignará como siempre, y los otros dos se introducirán en la tupla de args, en el orden en que se recibieron.

Usando ** kwargs al escribir funciones

Puede definir una función que tome un número arbitrario de argumentos de palabra clave (nombrados) usando la estrella doble `**` antes del nombre de un parámetro:

```
def print_kwargs(**kwargs):
    print(kwargs)
```

Al llamar al método, Python construirá un diccionario de todos los argumentos de palabras clave y lo pondrá a disposición en el cuerpo de la función:

```
print_kwargs(a="two", b=3)
# prints: "{a: "two", b=3}"
```

Tenga en cuenta que el parámetro ** kwargs en la definición de la función siempre debe ser el último parámetro, y solo coincidirá con los argumentos que se pasaron después de los anteriores.

```
def example(a, **kw):
    print kw

example(a=2, b=3, c=4) # => {'b': 3, 'c': 4}
```

Dentro del cuerpo de la función, los `kwargs` se manipulan de la misma manera que un diccionario; para acceder a elementos individuales en `kwargs` , solo tienes que recorrerlos como lo harías con un diccionario normal:

```
def print_kwargs(**kwargs):
    for key in kwargs:
        print("key = {0}, value = {1}".format(key, kwargs[key]))
```

Ahora, al llamar a `print_kwargs(a="two", b=1)` muestra el siguiente resultado:

```
print_kwargs(a = "two", b = 1)
key = a, value = "two"
key = b, value = 1
```

Usando * args al llamar a funciones

Un caso de uso común para `*args` en una definición de función es delegar el procesamiento a una función envuelta o heredada. Un ejemplo típico podría estar en el método `__init__` una clase

```
class A(object):
    def __init__(self, b, c):
        self.y = b
        self.z = c

class B(A):
    def __init__(self, a, *args, **kwargs):
        super(B, self).__init__(*args, **kwargs)
        self.x = a
```

Aquí, el `a` parámetro es procesado por la clase niño después de todos los otros argumentos (posicionales y de palabras clave) son pasados a - y procesados por - la clase base.

Por ejemplo:

```
b = B(1, 2, 3)
b.x  # 1
b.y  # 2
b.z  # 3
```

Lo que sucede aquí es que la función de clase `B` `__init__` ve los argumentos `1, 2, 3` . Sabe que necesita tomar un argumento posicional (`a`), por lo que toma el primer argumento pasado en (`1`), por lo que en el alcance de la función `a == 1` .

A continuación, ve que necesita tomar un número arbitrario de argumentos posicionales (`*args`), por lo que toma el resto de los argumentos posicionales pasados en (`1, 2`) y los mete en `*args` . Ahora (en el ámbito de la función) `args == [2, 3]` .

Luego, llama a la función `__init__` clase `A` con `*args` . Python ve el `*` delante de args y "desempaqueta" la lista en argumentos. En este ejemplo, cuando la clase `B` 's `__init__` llamadas de función de clase `A` ' s `__init__` función, será pasado los argumentos `2, 3` (es decir, `A(2, 3)`).

Finalmente, establece su propia propiedad `x` en el primer argumento posicional `a` , que es igual a `1` .

Usando ** kwargs al llamar a funciones

Puede usar un diccionario para asignar valores a los parámetros de la función; usando los parámetros nombre como claves en el diccionario y el valor de estos argumentos enlazados a cada clave:

```
def test_func(arg1, arg2, arg3): # Usual function with three arguments
   print("arg1: %s" % arg1)
   print("arg2: %s" % arg2)
   print("arg3: %s" % arg3)

# Note that dictionaries are unordered, so we can switch arg2 and arg3. Only the names matter.
kwargs = {"arg3": 3, "arg2": "two"}

# Bind the first argument (ie. arg1) to 1, and use the kwargs dictionary to bind the others
test_var_args_call(1, **kwargs)
```

Usando * args al llamar a funciones

El efecto de usar el operador `*` en un argumento al llamar a una función es el de desempaquetar la lista o un argumento de tupla

```
def print_args(arg1, arg2):
    print(str(arg1) + str(arg2))

a = [1,2]
b = tuple([3,4])

print_args(*a)
# 12
print_args(*b)
# 34
```

Tenga en cuenta que la longitud del argumento destacado debe ser igual al número de argumentos de la función.

Un lenguaje común en Python es usar el operador de desempaquetado `*` con la función `zip` para revertir sus efectos:

```
a = [1,3,5,7,9]
b = [2,4,6,8,10]

zipped = zip(a,b)
# [(1,2), (3,4), (5,6), (7,8), (9,10)]

zip(*zipped)
```

```
# (1,3,5,7,9), (2,4,6,8,10)
```

Argumentos solo de palabra clave y requeridos de palabra clave

Python 3 le permite definir argumentos de función que solo pueden asignarse por palabra clave, incluso sin valores predeterminados. Esto se hace usando star * para consumir parámetros posicionales adicionales sin configurar los parámetros de palabras clave. Todos los argumentos después del * son argumentos de palabra clave solamente (es decir, no posicionales). Tenga en cuenta que si los argumentos de solo palabras clave no tienen un valor predeterminado, todavía son necesarios al llamar a la función.

```
def print_args(arg1, *args, keyword_required, keyword_only=True):
    print("first positional arg: {}".format(arg1))
    for arg in args:
        print("another positional arg: {}".format(arg))
    print("keyword_required value: {}".format(keyword_required))
    print("keyword_only value: {}".format(keyword_only))

print(1, 2, 3, 4) # TypeError: print_args() missing 1 required keyword-only argument:
'keyword_required'
print(1, 2, 3, keyword_required=4)
# first positional arg: 1
# another positional arg: 2
# another positional arg: 3
# keyword_required value: 4
# keyword_only value: True
```

Poblando los valores kwarg con un diccionario

```
def foobar(foo=None, bar=None):
    return "{}{}".format(foo, bar)

values = {"foo": "foo", "bar": "bar"}

foobar(**values) # "foobar"
```

** kwargs y valores por defecto

Para usar los valores por defecto con ** kwargs

```
def fun(**kwargs):
    print kwargs.get('value', 0)

fun()
# print 0
fun(value=1)
# print 1
```

Lea * args y ** kwargs en línea: https://riptutorial.com/es/python/topic/2475/--args-y----kwargs

Capítulo 3: Acceso a la base de datos

Observaciones

Python puede manejar muchos tipos diferentes de bases de datos. Para cada uno de estos tipos existe una API diferente. Así que fomente la similitud entre esas diferentes API, se ha introducido PEP 249.

> Esta API se ha definido para fomentar la similitud entre los módulos de Python que se utilizan para acceder a las bases de datos. Al hacer esto, esperamos lograr una consistencia que conduzca a módulos más fáciles de entender, código que generalmente es más portátil en las bases de datos y un mayor alcance de la conectividad de base de datos de Python. PEP-249

Examples

Accediendo a la base de datos MySQL usando MySQLdb

Lo primero que debe hacer es crear una conexión a la base de datos utilizando el método de conexión. Después de eso, necesitará un cursor que operará con esa conexión.

Utilice el método de ejecución del cursor para interactuar con la base de datos y, de vez en cuando, confirme los cambios utilizando el método de confirmación del objeto de conexión.

Una vez hecho todo, no olvides cerrar el cursor y la conexión.

Aquí hay una clase Dbconnect con todo lo que necesitas.

```
import MySQLdb

class Dbconnect(object):

    def __init__(self):

        self.dbconection = MySQLdb.connect(host='host_example',
                                           port=int('port_example'),
                                           user='user_example',
                                           passwd='pass_example',
                                           db='schema_example')
        self.dbcursor = self.dbconection.cursor()

    def commit_db(self):
        self.dbconection.commit()

    def close_db(self):
        self.dbcursor.close()
        self.dbconection.close()
```

Interactuar con la base de datos es simple. Después de crear el objeto, simplemente use el método de ejecución.

```
db = Dbconnect()
db.dbcursor.execute('SELECT * FROM %s' % 'table_example')
```

Si desea llamar a un procedimiento almacenado, use la siguiente sintaxis. Tenga en cuenta que la lista de parámetros es opcional.

```
db = Dbconnect()
db.callproc('stored_procedure_name', [parameters] )
```

Una vez que se realiza la consulta, puede acceder a los resultados de varias maneras. El objeto del cursor es un generador que puede obtener todos los resultados o ser enlazado.

```
results = db.dbcursor.fetchall()
for individual_row in results:
    first_field = individual_row[0]
```

Si quieres un loop usando directamente el generador:

```
for individual_row in db.dbcursor:
    first_field = individual_row[0]
```

Si desea confirmar los cambios en la base de datos:

```
db.commit_db()
```

Si quieres cerrar el cursor y la conexión:

```
db.close_db()
```

SQLite

SQLite es una base de datos ligera, basada en disco. Dado que no requiere un servidor de base de datos separado, a menudo se usa para hacer prototipos o para aplicaciones pequeñas que a menudo usan un solo usuario o un usuario en un momento dado.

```
import sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

c.execute("CREATE TABLE user (name text, age integer)")

c.execute("INSERT INTO user VALUES ('User A', 42)")
c.execute("INSERT INTO user VALUES ('User B', 43)")

conn.commit()

c.execute("SELECT * FROM user")
print(c.fetchall())

conn.close()
```

El código anterior se conecta a la base de datos almacenada en el archivo llamado `users.db` , creando primero el archivo si aún no existe. Puede interactuar con la base de datos a través de sentencias de SQL.

El resultado de este ejemplo debe ser:

```
[(u'User A', 42), (u'User B', 43)]
```

La sintaxis de SQLite: un análisis en profundidad

Empezando

1. Importar el módulo sqlite usando

   ```
   >>> import sqlite3
   ```

2. Para usar el módulo, primero debe crear un objeto de conexión que represente la base de datos. Aquí los datos se almacenarán en el archivo example.db:

   ```
   >>> conn = sqlite3.connect('users.db')
   ```

   Alternativamente, también puede proporcionar el nombre especial `:memory:` para crear una base de datos temporal en la RAM, de la siguiente manera:

   ```
   >>> conn = sqlite3.connect(':memory:')
   ```

3. Una vez que tenga una `Connection` , puede crear un objeto `Cursor` y llamar a su método `execute()` para ejecutar comandos SQL:

   ```
   c = conn.cursor()

   # Create table
   c.execute('''CREATE TABLE stocks
                (date text, trans text, symbol text, qty real, price real)''')

   # Insert a row of data
   c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

   # Save (commit) the changes
   conn.commit()

   # We can also close the connection if we are done with it.
   # Just be sure any changes have been committed or they will be lost.
   conn.close()
   ```

Atributos importantes y funciones de `Connection`

1. `isolation_level`

   Es un atributo utilizado para obtener o establecer el nivel de aislamiento actual. Ninguno para el modo de confirmación automática o uno de `DEFERRED` , `IMMEDIATE` o `EXCLUSIVE` .

2. `cursor`

   El objeto del cursor se utiliza para ejecutar comandos y consultas SQL.

3. `commit()`

   Confirma la transacción actual.

4. `rollback()`

   Deshace los cambios realizados desde la llamada anterior a `commit()`

5. `close()`

   Cierra la conexión de la base de datos. No llama a `commit()` automáticamente. Si se llama a `close()` sin llamar primero a `commit()` (suponiendo que no esté en modo de `commit()` automática), se perderán todos los cambios realizados.

6. `total_changes`

   Un atributo que registra el número total de filas modificadas, eliminadas o insertadas desde que se abrió la base de datos.

7. `execute` , `executemany` y `executescript`

   Estas funciones se realizan de la misma manera que las del objeto cursor. Este es un atajo ya que llamar a estas funciones a través del objeto de conexión da como resultado la creación de un objeto de cursor intermedio y llama al método correspondiente del objeto de cursor

8. `row_factory`

   Puede cambiar este atributo a un llamable que acepte el cursor y la fila original como una tupla y devolverá la fila del resultado real.

```
def dict_factory(cursor, row):
    d = {}
    for i, col in enumerate(cursor.description):
        d[col[0]] = row[i]
    return d

conn = sqlite3.connect(":memory:")
conn.row_factory = dict_factory
```

Funciones importantes del `Cursor`

1. `execute(sql[, parameters])`

Ejecuta una *sola* sentencia SQL. La declaración SQL puede estar parametrizada (es decir, marcadores de posición en lugar de literales de SQL). El módulo sqlite3 admite dos tipos de marcadores de posición: ¿signos de interrogación `?` ("Estilo de qmark") y marcadores de posición con `:name` ("estilo con nombre").

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.execute("create table people (name, age)")

who = "Sophia"
age = 37
# This is the qmark style:
cur.execute("insert into people values (?, ?)",
            (who, age))

# And this is the named style:
cur.execute("select * from people where name=:who and age=:age",
            {"who": who, "age": age})  # the keys correspond to the placeholders in SQL

print(cur.fetchone())
```

Cuidado: no utilice `%s` para insertar cadenas en los comandos SQL, ya que puede hacer que su programa sea vulnerable a un ataque de inyección de SQL (consulte Inyección de SQL).

2. `executemany(sql, seq_of_parameters)`

   Ejecuta un comando SQL contra todas las secuencias de parámetros o asignaciones encontradas en la secuencia sql. El módulo sqlite3 también permite usar un iterador para producir parámetros en lugar de una secuencia.

```
L = [(1, 'abcd', 'dfj', 300),    # A list of tuples to be inserted into the database
     (2, 'cfgd', 'dyfj', 400),
     (3, 'sdd', 'dfjh', 300.50)]

conn = sqlite3.connect("test1.db")
conn.execute("create table if not exists book (id int, name text, author text, price
real)")
conn.executemany("insert into book values (?, ?, ?, ?)", L)

for row in conn.execute("select * from book"):
    print(row)
```

   También puede pasar objetos de iterador como un parámetro a muchos, y la función se repetirá sobre cada tupla de valores que devuelve el iterador. El iterador debe devolver una tupla de valores.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
```

```
        return self

    def __next__(self):            # (use next(self) for Python 2)
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),)

conn = sqlite3.connect("abc.db")
cur = conn.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

rows = cur.execute("select c from characters")
for row in rows:
    print(row[0]),
```

3. `executescript(sql_script)`

   Este es un método de conveniencia no estándar para ejecutar varias declaraciones SQL a la vez. Primero emite una instrucción `COMMIT` , luego ejecuta el script SQL que obtiene como parámetro.

   `sql_script` puede ser una instancia de `str` o `bytes` .

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.executescript("""
     create table person(
         firstname,
         lastname,
         age
     );

     create table book(
         title,
         author,
         published
     );

     insert into book(title, author, published)
     values (
         'Dirk Gently''s Holistic Detective Agency',
         'Douglas Adams',
         1987
     );
     """)
```

   El siguiente conjunto de funciones se usa junto con las `SELECT` en SQL. Para recuperar datos después de ejecutar una instrucción `SELECT` , puede tratar el cursor como un iterador, llamar al método `fetchone()` del cursor para recuperar una sola fila coincidente, o llamar a `fetchall()` para obtener una lista de las filas correspondientes.

   Ejemplo de la forma iterador:

```
import sqlite3
stocks = [('2006-01-05', 'BUY', 'RHAT', 100, 35.14),
          ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.0),
          ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
conn = sqlite3.connect(":memory:")
conn.execute("create table stocks (date text, buysell text, symb text, amount int, price
real)")
conn.executemany("insert into stocks values (?, ?, ?, ?, ?)", stocks)
cur = conn.cursor()

for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
    print(row)

# Output:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

4. `fetchone()`

   Obtiene la siguiente fila de un conjunto de resultados de consulta, devolviendo una secuencia única o Ninguno cuando no hay más datos disponibles.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
i = cur.fetchone()
while(i):
    print(i)
    i = cur.fetchone()

# Output:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

5. `fetchmany(size=cursor.arraysize)`

   Obtiene el siguiente conjunto de filas de un resultado de consulta (especificado por tamaño), devolviendo una lista. Si se omite el tamaño, fetchmany devuelve una sola fila. Se devuelve una lista vacía cuando no hay más filas disponibles.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchmany(2))

# Output:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)]
```

6. `fetchall()`

   Obtiene todas las filas (restantes) de un resultado de consulta, devolviendo una lista.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchall())
```

```
# Output:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
('2006-04-06', 'SELL', 'IBM', 500, 53.0), ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
```

Tipos de datos SQLite y Python

SQLite admite de forma nativa los siguientes tipos: NULL, INTEGER, REAL, TEXT, BLOB.

Así es como se convierten los tipos de datos al pasar de SQL a Python o viceversa.

```
None     <->     NULL
int      <->     INTEGER/INT
float    <->     REAL/FLOAT
str      <->     TEXT/VARCHAR(n)
bytes    <->     BLOB
```

Acceso a la base de datos PostgreSQL usando psycopg2

psycopg2 es el adaptador de base de datos PostgreSQL más popular que es ligero y eficiente. Es la implementación actual del adaptador PostgreSQL.

> Sus características principales son la implementación completa de la especificación Python DB API 2.0 y la seguridad de subprocesos (varios subprocesos pueden compartir la misma conexión)

Estableciendo una conexión a la base de datos y creando una tabla.

```
import psycopg2

# Establish a connection to the database.
# Replace parameter values with database credentials.
conn = psycopg2.connect(database="testpython",
                        user="postgres",
                        host="localhost",
                        password="abc123",
                        port="5432")

# Create a cursor. The cursor allows you to execute database queries.
cur = conn.cursor()

# Create a table. Initialise the table name, the column names and data type.
cur.execute("""CREATE TABLE FRUITS (
                    id          INT ,
                    fruit_name  TEXT,
                    color       TEXT,
                    price       REAL
            )""")
conn.commit()
conn.close()
```

Insertando datos en la tabla:

```
# After creating the table as shown above, insert values into it.
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Apples', 'green', 1.00)""")

cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Bananas', 'yellow', 0.80)""")
```

Recuperando datos de la tabla:

```
# Set up a query and execute it
cur.execute("""SELECT id, fruit_name, color, price
             FROM fruits""")

# Fetch the data
rows = cur.fetchall()

# Do stuff with the data
for row in rows:
    print "ID = {} ".format(row[0])
    print "FRUIT NAME = {}".format(row[1])
    print("COLOR = {}".format(row[2]))
    print("PRICE = {}".format(row[3]))
```

La salida de lo anterior sería:

```
ID = 1
NAME = Apples
COLOR = green
PRICE = 1.0

ID = 2
NAME = Bananas
COLOR = yellow
PRICE = 0.8
```

Y así, ahí tienes, ¡ahora sabes la mitad de todo lo que necesitas saber sobre ***psycopg2*** ! :)

Base de datos Oracle

Pre-requisitos:

- Paquete cx_Oracle - Vea aquí para todas las versiones
- Cliente instantáneo de Oracle - Para Windows x64 , Linux x64

Preparar:

- Instala el paquete cx_Oracle como:

  ```
  sudo rpm -i <YOUR_PACKAGE_FILENAME>
  ```

- Extraiga el cliente instantáneo de Oracle y establezca las variables de entorno como:

```
ORACLE_HOME=<PATH_TO_INSTANTCLIENT>
PATH=$ORACLE_HOME:$PATH
LD_LIBRARY_PATH=<PATH_TO_INSTANTCLIENT>:$LD_LIBRARY_PATH
```

Creando una conexión:

```
import cx_Oracle

class OraExec(object):
    _db_connection = None
    _db_cur = None

    def __init__(self):
        self._db_connection =
            cx_Oracle.connect('<USERNAME>/<PASSWORD>@<HOSTNAME>:<PORT>/<SERVICE_NAME>')
        self._db_cur = self._db_connection.cursor()
```

Obtener la versión de la base de datos:

```
ver = con.version.split(".")
print ver
```

Salida de muestra: ['12', '1', '0', '2', '0']

Ejecutar consulta: SELECCIONAR

```
_db_cur.execute("select * from employees order by emp_id")
for result in _db_cur:
    print result
```

La salida será en tuplas de Python:

(10, 'SYSADMIN', 'IT-INFRA', 7)

(23, 'HR ASSOCIATE', 'RECURSOS HUMANOS', 6)

Ejecutar consulta: INSERTAR

```
_db_cur.execute("insert into employees(emp_id, title, dept, grade)
                values (31, 'MTS', 'ENGINEERING', 7)
_db_connection.commit()
```

Cuando realiza operaciones de inserción / actualización / eliminación en una base de datos Oracle, los cambios solo están disponibles dentro de su sesión hasta `commit` se emita la `commit` . Cuando los datos actualizados se confirman en la base de datos, están disponibles para otros usuarios y sesiones.

Ejecutar consulta: INSERTAR utilizando variables Bind

Referencia

Las variables de vinculación le permiten volver a ejecutar sentencias con nuevos valores, sin la sobrecarga de volver a analizar la sentencia. Las variables de enlace mejoran la reutilización del código y pueden reducir el riesgo de ataques de inyección de SQL.

```
rows = [ (1, "First" ),
     (2, "Second" ),
     (3, "Third" ) ]
_db_cur.bindarraysize = 3
_db_cur.setinputsizes(int, 10)
_db_cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)
_db_connection.commit()
```

Conexión cercana:

```
_db_connection.close()
```

El método close () cierra la conexión. Cualquier conexión no cerrada explícitamente se liberará automáticamente cuando finalice el script.

Conexión

Creando una conexión

Según PEP 249, la conexión a una base de datos debe establecerse mediante un constructor `connect()` , que devuelve un objeto `Connection` . Los argumentos para este constructor son dependientes de la base de datos. Consulte los temas específicos de la base de datos para los argumentos relevantes.

```
import MyDBAPI

con = MyDBAPI.connect(*database_dependent_args)
```

Este objeto de conexión tiene cuatro métodos:

1: cerrar

```
con.close()
```

Cierra la conexión al instante. Tenga en cuenta que la conexión se cierra automáticamente si se llama al método `Connection.__del___` . Todas las transacciones pendientes se revertirán implícitamente.

2: cometer

```
con.commit()
```

Se compromete cualquier transacción pendiente a la base de datos.

3: retroceso

```
con.rollback()
```

Retrocede al inicio de cualquier transacción pendiente. En otras palabras: esto cancela cualquier transacción no confirmada a la base de datos.

4: cursor

```
cur = con.cursor()
```

Devuelve un objeto `Cursor` . Esto se utiliza para hacer transacciones en la base de datos.

Usando sqlalchemy

Para usar sqlalchemy para la base de datos:

```
from sqlalchemy import create_engine
from sqlalchemy.engine.url import URL


url = URL(drivername='mysql',
          username='user',
          password='passwd',
          host='host',
          database='db')

engine = create_engine(url)  # sqlalchemy engine
```

Ahora se puede usar este motor: por ejemplo, con pandas para obtener marcos de datos directamente desde mysql

```
import pandas as pd

con = engine.connect()
dataframe = pd.read_sql(sql=query, con=con)
```

Lea Acceso a la base de datos en línea: https://riptutorial.com/es/python/topic/4240/acceso-a-la-base-de-datos

Capítulo 4: Acceso al código fuente y código de bytes de Python

Examples

Mostrar el bytecode de una función

El intérprete de Python compila el código a bytecode antes de ejecutarlo en la máquina virtual de Python (consulte también ¿Qué es el bytecode de python?) .

Aquí es cómo ver el código de bytes de una función de Python

```
import dis

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)

# Display the disassembled bytecode of the function.
dis.dis(fib)
```

La función `dis.dis` en el módulo dis devolverá un bytecode descompilado de la función que se le pasó.

Explorando el código objeto de una función.

CPython permite el acceso al objeto de código para un objeto de función.

El objeto `__code__` contiene el bytecode en bruto (`co_code`) de la función, así como otra información como constantes y nombres de variables.

```
def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)
```

Mostrar el código fuente de un objeto.

Objetos que no están incorporados

Para imprimir el código fuente de un objeto Python use `inspect` . Tenga en cuenta que esto no funcionará para los objetos incorporados ni para los objetos definidos de forma interactiva. Para estos necesitarás otros métodos explicados más adelante.

Aquí le mostramos cómo imprimir el código fuente del método `randint` desde el módulo `random` :

```
import random
import inspect

print(inspect.getsource(random.randint))
# Output:
#    def randint(self, a, b):
#        """Return random integer in range [a, b], including both end points.
#        """
#
#        return self.randrange(a, b+1)
```

Para simplemente imprimir la cadena de documentación

```
print(inspect.getdoc(random.randint))
# Output:
# Return random integer in range [a, b], including both end points.
```

Imprima la ruta completa del archivo donde se define el método `random.randint` :

```
print(inspect.getfile(random.randint))
# c:\Python35\lib\random.py
print(random.randint.__code__.co_filename) # equivalent to the above
# c:\Python35\lib\random.py
```

Objetos definidos interactivamente.

Si un objeto está definido de forma interactiva, `inspect` no puede proporcionar el código fuente, pero puede usar `dill.source.getsource` en `dill.source.getsource` lugar

```
# define a new function in the interactive shell
def add(a, b):
   return a + b
print(add.__code__.co_filename) # Output: <stdin>

import dill
print dill.source.getsource(add)
# def add(a, b):
      return a + b
```

Objetos incorporados

El código fuente de las funciones incorporadas de Python está escrito en **c** y solo se puede acceder a él consultando el código fuente de Python (alojado en Mercurial o descargable desde https://www.python.org/downloads/source/) .

```
print(inspect.getsource(sorted)) # raises a TypeError
type(sorted) # <class 'builtin_function_or_method'>
```

Lea Acceso al código fuente y código de bytes de Python en línea:

https://riptutorial.com/es/python/topic/4351/acceso-al-codigo-fuente-y-codigo-de-bytes-de-python

Capítulo 5: Acceso de atributo

Sintaxis

- `x.title # Accesses the title attribute using the dot notation`
- `x.title = "Hello World" # Sets the property of the title attribute using the dot notation`
- `@property # Used as a decorator before the getter method for properties`
- `@title.setter # Used as a decorator before the setter method for properties`

Examples

Acceso a atributos básicos utilizando la notación de puntos

Tomemos una clase de muestra.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

book1 = Book(title="Right Ho, Jeeves", author="P.G. Wodehouse")
```

En Python, puede acceder al *título* del atributo de la clase usando la notación de puntos.

```
>>> book1.title
'P.G. Wodehouse'
```

Si un atributo no existe, Python lanza un error:

```
>>> book1.series
Traceback  (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Book' object has no attribute 'series'
```

Setters, Getters & Properties

Por el bien de la encapsulación de datos, a veces desea tener un atributo cuyo valor proviene de otros atributos o, en general, qué valor se computará en el momento. La forma estándar de lidiar con esta situación es crear un método, llamado getter o setter.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

En el ejemplo anterior, es fácil ver qué sucede si creamos un nuevo libro que contiene un título y un autor. Si todos los libros que vamos a agregar a nuestra biblioteca tienen autores y títulos, podemos omitir a los captadores y definidores y usar la notación de puntos. Sin embargo,

supongamos que tenemos algunos libros que no tienen un autor y queremos que el autor sea "Desconocido". O si tienen varios autores y planeamos devolver una lista de autores.

En este caso, podemos crear un getter y un setter para el atributo de *autor* .

```
class P:
    def __init__(self,title,author):
        self.title = title
        self.setAuthor(author)

    def get_author(self):
        return self.author

    def set_author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Este esquema no es recomendable.

Una razón es que hay un problema: supongamos que hemos diseñado nuestra clase con el atributo público y sin métodos. La gente ya lo ha usado mucho y ha escrito un código como este:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
```

Ahora tenemos un problema. ¡Porque el *autor* no es un atributo! Python ofrece una solución a este problema llamado propiedades. Un método para obtener propiedades está decorado con la propiedad @ antes de su encabezado. El método que queremos que funcione como configurador está decorado con @ attributeName.setter anterior.

Teniendo esto en cuenta, ahora tenemos nuestra nueva clase actualizada.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    @property
    def author(self):
        return self.__author

    @author.setter
    def author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Tenga en cuenta que normalmente Python no le permite tener varios métodos con el mismo nombre y diferente número de parámetros. Sin embargo, en este caso Python lo permite debido a los decoradores utilizados.

Si probamos el código:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
>>> book.author
Unknown
```

Lea Acceso de atributo en línea: https://riptutorial.com/es/python/topic/4392/acceso-de-atributo

Capítulo 6: agrupar por()

Introducción

En Python, el método `itertools.groupby()` permite a los desarrolladores agrupar los valores de una clase iterable basándose en una propiedad específica en otro conjunto de valores iterables.

Sintaxis

- itertools.groupby (iterable, key = None o alguna función)

Parámetros

Parámetro	Detalles
iterable	Cualquier pitón iterable
llave	Función (criterios) sobre la cual agrupar lo iterable.

Observaciones

groupby () es complicado pero una regla general a tener en cuenta al usarlo es la siguiente:

Ordene siempre los elementos que desea agrupar con la misma clave que desea utilizar para agrupar

Se recomienda que el lector revise la documentación aquí y vea cómo se explica utilizando una definición de clase.

Examples

Ejemplo 1

Di que tienes la cuerda

```
s = 'AAAABBBCCDAABBB'
```

y te gustaría dividirlo para que todas las 'A' estén en una lista y así con todas las 'B' y 'C', etc. Podrías hacer algo como esto

```
s = 'AAAABBBCCDAABBB'
s_dict = {}
for i in s:
    if i not in s_dict.keys():
```

```
        s_dict[i] = [i]
    else:
        s_dict[i].append(i)
s_dict
```

Resultados en

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'],
 'B': ['B', 'B', 'B', 'B', 'B', 'B'],
 'C': ['C', 'C'],
 'D': ['D']}
```

Pero para un conjunto de datos de gran tamaño, estaría acumulando estos elementos en la memoria. Aquí es donde entra groupby ()

Podríamos obtener el mismo resultado de una manera más eficiente haciendo lo siguiente

```
# note that we get a {key : value} pair for iterating over the items just like in python
dictionary
from itertools import groupby
s = 'AAAABBBCCDAABBB'
c = groupby(s)

dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Resultados en

```
{'A': ['A', 'A'], 'B': ['B', 'B', 'B'], 'C': ['C', 'C'], 'D': ['D']}
```

Observe que el número de 'A's en el resultado cuando usamos group by es menor que el número real de' A's en la cadena original. Podemos evitar esa pérdida de información ordenando los elementos en s antes de pasarlos a c como se muestra a continuación

```
c = groupby(sorted(s))

dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Resultados en

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'], 'B': ['B', 'B', 'B', 'B', 'B', 'B'], 'C': ['C', 'C'],
'D': ['D']}
```

Ahora tenemos todas nuestras 'A's.

Ejemplo 2

Este ejemplo ilustra cómo se elige la clave predeterminada si no especificamos ninguna

```
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('persons', 'man', 'woman')])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Resultados en

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Observe aquí que la tupla en su conjunto cuenta como una clave en esta lista

Ejemplo 3

Note en este ejemplo que mulato y camello no aparecen en nuestro resultado. Sólo se muestra el último elemento con la clave especificada. El último resultado para c en realidad borra dos resultados anteriores. Pero mira la nueva versión donde tengo los datos ordenados primero en la misma clave.

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
c = groupby(list_things, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Resultados en

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Versión ordenada

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
sorted_list = sorted(list_things, key = lambda x: x[0])
print(sorted_list)
print()
```

```
c = groupby(sorted_list, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Resultados en

```
['cow', 'cat', 'camel', 'dog', 'donkey', 'goat', 'mulato', 'mongoose', 'malloo', ('persons',
'man', 'woman'), 'wombat']

{'c': ['cow', 'cat', 'camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mulato', 'mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Ejemplo 4

En este ejemplo, vemos lo que sucede cuando usamos diferentes tipos de iterable.

```
things = [("animal", "bear"), ("animal", "duck"), ("plant", "cactus"), ("vehicle", "harley"),
\
          ("vehicle", "speed boat"), ("vehicle", "school bus")]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Resultados en

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
 'plant': [('plant', 'cactus')],
 'vehicle': [('vehicle', 'harley'),
  ('vehicle', 'speed boat'),
  ('vehicle', 'school bus')]}
```

Este ejemplo a continuación es esencialmente el mismo que el de arriba. La única diferencia es que he cambiado todas las tuplas a listas.

```
things = [["animal", "bear"], ["animal", "duck"], ["vehicle", "harley"], ["plant", "cactus"],
\
          ["vehicle", "speed boat"], ["vehicle", "school bus"]]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Resultados

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
 'plant': [['plant', 'cactus']],
 'vehicle': [['vehicle', 'harley'],
  ['vehicle', 'speed boat'],
  ['vehicle', 'school bus']]}
```

Lea agrupar por() en línea: https://riptutorial.com/es/python/topic/8690/agrupar-por--

Capítulo 7: Alcance variable y vinculante

Sintaxis

- global a, b, c
- no local a, b
- x = algo # se une a x
- (x, y) = algo # une x y y
- x + = algo # se une a x. Del mismo modo para todos los demás "op ="
- del x # se une a x
- para x en algo: # se une a x
- con algo como x: # une x
- excepto Excepción como ex: # binds ex inside block

Examples

Variables globales

En Python, las variables dentro de las funciones se consideran locales si y solo si aparecen en el lado izquierdo de una instrucción de asignación, o alguna otra ocurrencia de enlace; de lo contrario, tal enlace se busca en las funciones adjuntas, hasta el alcance global. Esto es cierto incluso si la instrucción de asignación nunca se ejecuta.

```
x = 'Hi'

def read_x():
    print(x)   # x is just referenced, therefore assumed global

read_x()       # prints Hi

def read_y():
    print(y)   # here y is just referenced, therefore assumed global

read_y()       # NameError: global name 'y' is not defined

def read_y():
    y = 'Hey'  # y appears in an assignment, therefore it's local
    print(y)   # will find the local y

read_y()       # prints Hey

def read_x_local_fail():
    if False:
        x = 'Hey'  # x appears in an assignment, therefore it's local
    print(x)   # will look for the _local_ z, which is not assigned, and will not be found

read_x_local_fail()   # UnboundLocalError: local variable 'x' referenced before assignment
```

Normalmente, una asignación dentro de un ámbito sombreará las variables externas del mismo nombre:

```
x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
change_local_x()  # prints Bye
print(x)  # prints Hi
```

Declarar un nombre `global` significa que, para el resto del alcance, cualquier asignación al nombre ocurrirá en el nivel superior del módulo:

```
x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # prints Bye
print(x)  # prints Bye
```

La palabra clave `global` significa que las asignaciones se realizarán en el nivel superior del módulo, no en el nivel superior del programa. Otros módulos seguirán necesitando el acceso puntual habitual a las variables dentro del módulo.

Para resumir: para saber si una variable `x` es local a una función, debe leer la función *completa* :

1. Si has encontrado `global x` , entonces `x` es una variable **global**
2. Si ha encontrado `nonlocal x` , entonces `x` pertenece a una función que lo encierra, y no es local ni global
3. Si ha encontrado `x = 5` o `for x in range(3)` o algún otro enlace, entonces `x` es una variable **local**
4. De lo contrario, `x` pertenece a algún ámbito adjunto (ámbito de función, ámbito global o elementos incorporados)

Variables locales

Si un nombre está *enlazado* dentro de una función, por defecto es accesible solo dentro de la función:

```
def foo():
    a = 5
    print(a) # ok

print(a) #  NameError: name 'a' is not defined
```

Las construcciones de flujo de control no tienen impacto en el alcance (con la excepción de `except`), pero el acceso a la variable que aún no se asignó es un error:

```
def foo():
    if True:
        a = 5
```

```
    print(a) # ok

b = 3
def bar():
    if False:
        b = 5
    print(b) # UnboundLocalError: local variable 'b' referenced before assignment
```

Las operaciones de enlace comunes son asignaciones, `for` bucles y asignaciones aumentadas, como `a += 5`

Variables no locales

Python 3.x 3.0

Python 3 agregó una nueva palabra clave llamada **no local** . La palabra clave no local agrega una anulación de ámbito al ámbito interno. Puedes leer todo sobre esto en PEP 3104 . Esto se ilustra mejor con un par de ejemplos de código. Uno de los ejemplos más comunes es crear una función que puede incrementar:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

Si intenta ejecutar este código, recibirá un **UnboundLocalError** porque se hace referencia a la variable **num** antes de que se asigne en la función más interna. Añadamos nonlocal a la mezcla:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

Básicamente, `nonlocal` le permitirá asignar variables en un ámbito externo, pero no global. Por lo tanto, no puede usar el modo no `nonlocal` en nuestra función de `counter` porque entonces intentaría asignarlo a un alcance global. `SyntaxError` y obtendrá rápidamente un `SyntaxError` . En su lugar, debe utilizar `nonlocal` en una función anidada.

(Tenga en cuenta que la funcionalidad que se presenta aquí se implementa mejor utilizando generadores).

Ocurrencia vinculante

```
x = 5
x += 7
for x in iterable: pass
```

Cada una de las afirmaciones anteriores es una *ocurrencia de enlace* : `x` se une al objeto denotado por `5` . Si esta declaración aparece dentro de una función, entonces `x` será función local de forma predeterminada. Consulte la sección "Sintaxis" para obtener una lista de declaraciones vinculantes.

Las funciones omiten el alcance de la clase al buscar nombres

Las clases tienen un alcance local durante la definición, pero las funciones dentro de la clase no usan ese alcance cuando buscan nombres. Debido a que las lambdas son funciones, y las comprensiones se implementan utilizando el alcance de la función, esto puede llevar a un comportamiento sorprendente.

```
a = 'global'

class Fred:
    a = 'class'  # class scope
    b = (a for i in range(10))  # function scope
    c = [a for i in range(10)]  # function scope
    d = a  # class scope
    e = lambda: a  # function scope
    f = lambda a=a: a  # default argument uses class scope

    @staticmethod  # or @classmethod, or regular instance method
    def g():  # function scope
        return a

print(Fred.a)  # class
print(next(Fred.b))  # global
print(Fred.c[0])  # class in Python 2, global in Python 3
print(Fred.d)  # class
print(Fred.e())  # global
print(Fred.f())  # class
print(Fred.g()) # global
```

Los usuarios que no estén familiarizados con el funcionamiento de este ámbito pueden esperar que `b` , `c` y `e` impriman la `class` .

Desde PEP 227 :

> Los nombres en el alcance de la clase no son accesibles. Los nombres se resuelven en el ámbito de la función de cierre más interno. Si una definición de clase se produce en una cadena de ámbitos anidados, el proceso de resolución omite las definiciones de clase.

De la documentación de Python sobre denominación y encuadernación :

> El alcance de los nombres definidos en un bloque de clase se limita al bloque de clase; no se extiende a los bloques de código de los métodos; esto incluye

comprensiones y expresiones generadoras, ya que se implementan utilizando un ámbito de función. Esto significa que lo siguiente fallará:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

Este ejemplo utiliza referencias de esta respuesta de Martijn Pieters, que contiene un análisis más profundo de este comportamiento.

El comando del

Este comando tiene varias formas relacionadas pero distintas.

del v

Si `v` es una variable, el comando `del v` elimina la variable de su alcance. Por ejemplo:

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'f' is not defined
```

> Tenga en cuenta que `del` es una *ocurrencia de enlace* , lo que significa que a menos que se establezca explícitamente lo contrario (utilizando `nonlocal` o `global`), `del v` hará que `v` local al alcance actual. Si pretende eliminar `v` en un ámbito externo, use `nonlocal v` o `global v` en el mismo ámbito de la declaración `del v` .

En todo lo siguiente, la intención de un comando es un comportamiento predeterminado, pero el idioma no lo impone. Una clase puede estar escrita de una manera que invalida esta intención.

del v.name

Este comando activa una llamada a `v.__delattr__(name)` .

La intención es hacer que el `name` del atributo no esté disponible. Por ejemplo:

```
class A:
    pass

a = A()
a.x = 7
print(a.x) # out: 7
del a.x
print(a.x) # error: AttributeError: 'A' object has no attribute 'x'
```

del v[item]

Este comando activa una llamada a `v.__delitem__(item)` .

La intención es que el `item` no pertenezca a la asignación implementada por el objeto `v` . Por ejemplo:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) #  out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
```

`del v[a:b]`

Esto realmente llama `v.__delslice__(a, b)` .

La intención es similar a la descrita anteriormente, pero con cortes: rangos de elementos en lugar de un solo elemento. Por ejemplo:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) #  out: [0, 3, 4]
```

Véase también Recolección de basura # El comando del .

Ámbito local vs global

¿Cuáles son el alcance local y global?

Todos los variabes de Python a los que se puede acceder en algún punto del código se encuentran en *el ámbito local* o *global* .

La explicación es que el alcance local incluye todas las variables definidas en la función actual y el alcance global incluye variables definidas fuera de la función actual.

```
foo = 1  # global

def func():
    bar = 2  # local
    print(foo)  # prints variable foo from global scope
    print(bar)  # prints variable bar from local scope
```

Uno puede inspeccionar qué variables están en qué alcance. Las funciones integradas `locals()` y `globals()` devuelven todos los ámbitos como diccionarios.

```
foo = 1

def func():
    bar = 2
    print(globals().keys())  # prints all variable names in global scope
    print(locals().keys())  # prints all variable names in local scope
```

¿Qué pasa con los choques de nombre?

```
foo = 1

def func():
    foo = 2  # creates a new variable foo in local scope, global foo is not affected

    print(foo)  # prints 2

    # global variable foo still exists, unchanged:
    print(globals()['foo'])  # prints 1
    print(locals()['foo'])  # prints 2
```

Para modificar una variable global, use la palabra clave `global` :

```
foo = 1

def func():
    global foo
    foo = 2  # this modifies the global foo, rather than creating a local variable
```

El alcance se define para todo el cuerpo de la función!

Lo que significa es que una variable nunca será global para la mitad de la función y local después, o viceversa.

```
foo = 1

def func():
    # This function has a local variable foo, because it is defined down below.
    # So, foo is local from this point. Global foo is hidden.

    print(foo) # raises UnboundLocalError, because local foo is not yet initialized
    foo = 7
    print(foo)
```

Asimismo, el opuesto:

```
foo = 1

def func():
    # In this function, foo is a global variable from the begining

    foo = 7  # global foo is modified

    print(foo)  # 7
    print(globals()['foo'])  # 7

    global foo  # this could be anywhere within the function
    print(foo)  # 7
```

Funciones dentro de funciones

Puede haber muchos niveles de funciones anidadas dentro de las funciones, pero dentro de una función solo hay un ámbito local para esa función y el ámbito global. No hay ámbitos intermedios.

```
foo = 1

def f1():
    bar = 1

    def f2():
        baz = 2
        # here, foo is a global variable, baz is a local variable
        # bar is not in either scope
        print(locals().keys())  # ['baz']
        print('bar' in locals())  # False
        print('bar' in globals())  # False

    def f3():
        baz = 3
        print(bar)  # bar from f1 is referenced so it enters local scope of f3 (closure)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False

    def f4():
        bar = 4  # a new local bar which hides bar from local scope of f1
        baz = 4
        print(bar)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False
```

`global` vs `nonlocal` (solo Python 3)

Estas dos palabras clave se utilizan para obtener acceso de escritura a variables que no son locales a las funciones actuales.

La palabra clave `global` declara que un nombre debe tratarse como una variable global.

```
foo = 0  # global foo

def f1():
    foo = 1  # a new foo local in f1

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            foo = 3  # a new foo local in f3
            print(foo)  # 3
            foo = 30  # modifies local foo in f3 only

        def f4():
            global foo
            print(foo)  # 0
            foo = 100  # modifies global foo
```

Por otro lado, `nonlocal` (consulte Variables no locales), disponible en Python 3, toma una variable *local* de un ámbito de inclusión en el ámbito local de la función actual.

De la documentación de Python en `nonlocal` :

> La declaración no local hace que los identificadores enumerados se refieran a variables previamente vinculadas en el ámbito envolvente más cercano, excluyendo las globales.

Python 3.x 3.0

```
def f1():

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            nonlocal foo  # foo from f2, which is the nearest enclosing scope
            print(foo)  # 2
            foo = 20  # modifies foo from f2!
```

Lea Alcance variable y vinculante en línea: https://riptutorial.com/es/python/topic/263/alcance-variable-y-vinculante

Capítulo 8: Almohada

Examples

Leer archivo de imagen

```
from PIL import Image

im = Image.open("Image.bmp")
```

Convertir archivos a JPEG

```
from __future__ import print_function
import os, sys
from PIL import Image

for infile in sys.argv[1:]:
    f, e = os.path.splitext(infile)
    outfile = f + ".jpg"
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print("cannot convert", infile)
```

Lea Almohada en línea: https://riptutorial.com/es/python/topic/6841/almohada

Capítulo 9: Alternativas para cambiar la declaración de otros idiomas

Observaciones

No hay *ninguna* instrucción de cambio en python como opción de diseño de idioma. Ha habido un PEP (PEP-3103) que cubre el tema que ha sido rechazado.

Puede encontrar muchas listas de recetas sobre cómo hacer sus propias declaraciones de cambio en python, y aquí estoy tratando de sugerir las opciones más sensatas. Aquí hay algunos lugares para comprobar:

- http://stackoverflow.com/questions/60208/replacements-for-switch-statement-in-python
- http://code.activestate.com/recipes/269708-some-python-style-switches/
- http://code.activestate.com/recipes/410692-readable-switch-construction-without-lambdas-or-di/
- ...

Examples

Usa lo que el lenguaje ofrece: la construcción if / else.

Bueno, si quieres un constructo de `switch` / `case` , la forma más directa de hacerlo es usar el viejo y bueno `if` / `else` construye:

```
def switch(value):
    if value == 1:
        return "one"
    if value == 2:
        return "two"
    if value == 42:
        return "the answer to the question about life, the universe and everything"
    raise Exception("No case found!")
```

Puede parecer redundante, y no siempre bonito, pero esa es, con diferencia, la forma más eficiente de hacerlo, y hace el trabajo:

```
>>> switch(1)
one
>>> switch(2)
two
>>> switch(3)
…
Exception: No case found!
>>> switch(42)
the answer to the question about life the universe and everything
```

Usa un dictado de funciones.

Otra forma directa de hacerlo es crear un diccionario de funciones:

```
switch = {
    1: lambda: 'one',
    2: lambda: 'two',
    42: lambda: 'the answer of life the universe and everything',
}
```

a continuación, agrega una función predeterminada:

```
def default_case():
    raise Exception('No case found!')
```

y utiliza el método get del diccionario para obtener la función dado el valor para verificarlo y ejecutarlo. Si el valor no existe en el diccionario, entonces se ejecuta `default_case` .

```
>>> switch.get(1, default_case)()
one
>>> switch.get(2, default_case)()
two
>>> switch.get(3, default_case)()
…
Exception: No case found!
>>> switch.get(42, default_case)()
the answer of life the universe and everything
```

También puedes hacer un poco de azúcar sintáctico para que el interruptor se vea mejor:

```
def run_switch(value):
    return switch.get(value, default_case)()

>>> run_switch(1)
one
```

Usa la introspección de clase.

Puedes usar una clase para imitar la estructura del conmutador / caso. Lo siguiente es usar la introspección de una clase (usar la función `getattr()` que resuelve una cadena en un método enlazado en una instancia) para resolver la parte del "caso".

Luego, ese método de introspección se `__call__` método `__call__` para sobrecargar al operador `()` .

```
class SwitchBase:
    def switch(self, case):
        m = getattr(self, 'case_{}'.format(case), None)
        if not m:
            return self.default
        return m
```

```
    __call__ = switch
```

Luego, para que se vea mejor, subclasificamos la clase `SwitchBase` (pero podría hacerse en una clase), y ahí definimos todos los `case` como métodos:

```
class CustomSwitcher:
    def case_1(self):
        return 'one'

    def case_2(self):
        return 'two'

    def case_42(self):
        return 'the answer of life, the universe and everything!'

    def default(self):
        raise Exception('Not a case!')
```

así que finalmente podemos usarlo:

```
>>> switch = CustomSwitcher()
>>> print(switch(1))
one
>>> print(switch(2))
two
>>> print(switch(3))
…
Exception: Not a case!
>>> print(switch(42))
the answer of life, the universe and everything!
```

Usando un administrador de contexto

Otra forma, que es muy legible y elegante, pero mucho menos eficiente que una estructura if / else, es construir una clase como la siguiente, que leerá y almacenará el valor para comparar, exponerse dentro del contexto como un reclamo que devolverá verdadero si coincide con el valor almacenado:

```
class Switch:
    def __init__(self, value):
        self._val = value
    def __enter__(self):
        return self
    def __exit__(self, type, value, traceback):
        return False # Allows traceback to occur
    def __call__(self, cond, *mconds):
        return self._val in (cond,)+mconds
```

luego, definir los casos es casi una coincidencia con el constructo de `switch` / `case` real (expuesto dentro de una función a continuación, para que sea más fácil presumir):

```
def run_switch(value):
    with Switch(value) as case:
```

```
        if case(1):
            return 'one'
        if case(2):
            return 'two'
        if case(3):
            return 'the answer to the question about life, the universe and everything'
        # default
        raise Exception('Not a case!')
```

Entonces la ejecución sería:

```
>>> run_switch(1)
one
>>> run_switch(2)
two
>>> run_switch(3)
…
Exception: Not a case!
>>> run_switch(42)
the answer to the question about life, the universe and everything
```

Nota Bene :

- Esta solución se ofrece como el módulo de *conmutación* disponible en pypi .

Lea Alternativas para cambiar la declaración de otros idiomas en línea: https://riptutorial.com/es/python/topic/4268/alternativas-para-cambiar-la-declaracion-de-otros-idiomas

Capítulo 10: Ambiente Virtual Python - virtualenv

Introducción

Un entorno virtual ("virtualenv") es una herramienta para crear entornos aislados de Python. Mantiene las dependencias requeridas por los diferentes proyectos en lugares separados, mediante la creación de env de Python virtual para ellos. Resuelve el "proyecto A depende de la versión 2.xxx pero, el proyecto B necesita 2.xxx" dilema, y mantiene el directorio de paquetes de sitio global limpio y manejable.

"virtualenv" crea una carpeta que contiene todas las librerías y contenedores necesarios para usar los paquetes que un proyecto de Python necesitaría.

Examples

Instalación

Instale virtualenv a través de pip / (apt-get):

```
pip install virtualenv
```

O

```
apt-get install python-virtualenv
```

Nota: En caso de que tengas problemas con los permisos, usa sudo.

Uso

```
$ cd test_proj
```

Crear entorno virtual:

```
$ virtualenv test_proj
```

Para comenzar a utilizar el entorno virtual, debe activarse:

```
$ source test_project/bin/activate
```

Para salir de su virtualenv simplemente escriba "desactivar":

```
$ deactivate
```

Instala un paquete en tu Virtualenv

Si observa el directorio bin en su virtualenv, verá que easy_install se ha modificado para poner huevos y paquetes en el directorio de paquetes de sitio virtualenv. Para instalar una aplicación en su entorno virtual:

```
$ source test_project/bin/activate
$ pip install flask
```

En este momento, no tiene que usar sudo ya que todos los archivos se instalarán en el directorio local de virtualenv site-packages. Esto fue creado como su propia cuenta de usuario.

Otros comandos virtuales útiles

lsvirtualenv : Listar todos los entornos.

cdvirtualenv : navegue en el directorio del entorno virtual actualmente activado, para que pueda navegar por sus paquetes de sitios, por ejemplo.

cdsitepackages : como el anterior, pero directamente en el directorio de paquetes de sitio.

lssitepackages : muestra los contenidos del directorio site-packages.

Lea Ambiente Virtual Python - virtualenv en línea:
https://riptutorial.com/es/python/topic/9782/ambiente-virtual-python---virtualenv

Capítulo 11: Análisis de argumentos de línea de comandos

Introducción

La mayoría de las herramientas de línea de comandos se basan en argumentos pasados al programa en su ejecución. En lugar de solicitar una entrada, estos programas esperan que se establezcan datos o indicadores específicos (que se convierten en valores booleanos). Esto permite que tanto el usuario como otros programas ejecuten el archivo Python pasándole los datos a medida que se inician. Esta sección explica y demuestra la implementación y el uso de los argumentos de la línea de comandos en Python.

Examples

Hola mundo en argparse

El siguiente programa dice hola al usuario. Toma un argumento posicional, el nombre del usuario, y también se le puede decir el saludo.

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('name',
    help='name of user'
)

parser.add_argument('-g', '--greeting',
    default='Hello',
    help='optional alternate greeting'
)

args = parser.parse_args()

print("{greeting}, {name}!".format(
       greeting=args.greeting,
       name=args.name)
)
```

```
$ python hello.py --help
usage: hello.py [-h] [-g GREETING] name

positional arguments:
  name                  name of user

optional arguments:
  -h, --help            show this help message and exit
  -g GREETING, --greeting GREETING
                        optional alternate greeting
```

```
$ python hello.py world
Hello, world!
$ python hello.py John -g Howdy
Howdy, John!
```

Para más detalles por favor lea la documentación argparse .

Ejemplo básico con docopt.

docopt convierte el argumento de la línea de comando analizando en su cabeza. En lugar de analizar los argumentos, simplemente **escriba la cadena de uso** para su programa, y docopt **analiza la cadena de uso** y la utiliza para extraer los argumentos de la línea de comandos.

```
"""
Usage:
    script_name.py [-a] [-b] <path>

Options:
    -a            Print all the things.
    -b            Get more bees into the path.
"""
from docopt import docopt


if __name__ == "__main__":
    args = docopt(__doc__)
    import pprint; pprint.pprint(args)
```

Ejecuciones de muestra:

```
$ python script_name.py
Usage:
    script_name.py [-a] [-b] <path>
$ python script_name.py something
{'-a': False,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py something -a
{'-a': True,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py -b something -a
{'-a': True,
 '-b': True,
 '<path>': 'something'}
```

Estableciendo argumentos mutuamente excluyentes con argparse

Si quieres que dos o más argumentos sean mutuamente excluyentes. Puede utilizar la función `argparse.ArgumentParser.add_mutually_exclusive_group()` . En el siguiente ejemplo, pueden existir foo o bar, pero no ambos al mismo tiempo.

```
import argparse
```

```
parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-f", "--foo")
group.add_argument("-b", "--bar")
args = parser.parse_args()
print "foo = ", args.foo
print "bar = ", args.bar
```

Si intenta ejecutar el script especificando los argumentos `--foo` y `--bar` , el script se quejará con el mensaje a continuación.

```
error: argument -b/--bar: not allowed with argument -f/--foo
```

Usando argumentos de línea de comando con argv

Cada vez que se invoca un script de Python desde la línea de comandos, el usuario puede proporcionar **argumentos de línea de comandos** adicionales que se pasarán al script. Estos argumentos estarán disponibles para el programador de la variable del sistema `sys.argv` ("argv" es un nombre tradicional utilizado en la mayoría de los lenguajes de programación, y significa **"arg** ument **v** ector").

Por convención, el primer elemento de la lista `sys.argv` es el nombre de la secuencia de comandos de Python, mientras que el resto de los elementos son los tokens que el usuario pasa al invocar la secuencia de comandos.

```
# cli.py
import sys
print(sys.argv)

$ python cli.py
=> ['cli.py']

$ python cli.py fizz
=> ['cli.py', 'fizz']

$ python cli.py fizz buzz
=> ['cli.py', 'fizz', 'buzz']
```

Aquí hay otro ejemplo de cómo usar `argv` . Primero eliminamos el elemento inicial de sys.argv porque contiene el nombre del script. Luego combinamos el resto de los argumentos en una sola oración, y finalmente imprimimos esa oración antes del nombre del usuario que ha iniciado sesión (para que emule un programa de chat).

```
import getpass
import sys

words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

El algoritmo comúnmente utilizado cuando se analiza "manualmente" un número de argumentos no posicionales es iterar sobre la lista `sys.argv` . Una forma es repasar la lista y resaltar cada elemento de la misma:

```
# reverse and copy sys.argv
argv = reversed(sys.argv)
# extract the first element
arg = argv.pop()
# stop iterating when there's no more args to pop()
while len(argv) > 0:
    if arg in ('-f', '--foo'):
        print('seen foo!')
    elif arg in ('-b', '--bar'):
        print('seen bar!')
    elif arg in ('-a', '--with-arg'):
        arg = arg.pop()
        print('seen value: {}'.format(arg))
    # get the next value
    arg = argv.pop()
```

Mensaje de error del analizador personalizado con argparse

Puede crear mensajes de error del analizador de acuerdo con las necesidades de su script. Esto es a través de la función `argparse.ArgumentParser.error` . El siguiente ejemplo muestra la secuencia de comandos imprimiendo un uso y un mensaje de error a `stderr` cuando se `--foo` pero no `--bar` .

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-f", "--foo")
parser.add_argument("-b", "--bar")
args = parser.parse_args()
if args.foo and args.bar is None:
    parser.error("--foo requires --bar. You did not specify bar.")

print "foo =", args.foo
print "bar =", args.bar
```

Asumiendo que el nombre de su script es sample.py, y ejecutamos: `python sample.py --foo ds_in_fridge`

El guión se quejará con lo siguiente:

```
usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.
```

Agrupación conceptual de argumentos con argparse.add_argument_group ()

Cuando crea un argparse ArgumentParser () y ejecuta su programa con '-h', recibe un mensaje de uso automático que explica con qué argumentos puede ejecutar su software. De forma predeterminada, los argumentos posicionales y los argumentos condicionales están separados en dos categorías, por ejemplo, aquí hay un pequeño script (example.py) y el resultado cuando ejecuta `python example.py -h` .

```
import argparse
```

```
parser = argparse.ArgumentParser(description='Simple example')
parser.add_argument('name', help='Who to greet', default='World')
parser.add_argument('--bar_this')
parser.add_argument('--bar_that')
parser.add_argument('--foo_this')
parser.add_argument('--foo_that')
args = parser.parse_args()
```

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                      [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                      name

Simple example

positional arguments:
  name                  Who to greet

optional arguments:
  -h, --help            show this help message and exit
  --bar_this BAR_THIS
  --bar_that BAR_THAT
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

Hay algunas situaciones en las que desea separar sus argumentos en secciones conceptuales adicionales para ayudar a su usuario. Por ejemplo, es posible que desee tener todas las opciones de entrada en un grupo y todas las opciones de formato de salida en otro. El ejemplo anterior se puede ajustar para separar los `--foo_*` de los `--bar_*` así.

```
import argparse

parser = argparse.ArgumentParser(description='Simple example')
parser.add_argument('name', help='Who to greet', default='World')
# Create two argument groups
foo_group = parser.add_argument_group(title='Foo options')
bar_group = parser.add_argument_group(title='Bar options')
# Add arguments to those groups
foo_group.add_argument('--bar_this')
foo_group.add_argument('--bar_that')
bar_group.add_argument('--foo_this')
bar_group.add_argument('--foo_that')
args = parser.parse_args()
```

Que produce esta salida cuando se ejecuta `python example.py -h` :

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                      [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                      name

Simple example

positional arguments:
  name                  Who to greet

optional arguments:
  -h, --help            show this help message and exit
```

```
Foo options:
  --bar_this BAR_THIS
  --bar_that BAR_THAT

Bar options:
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

Ejemplo avanzado con docopt y docopt_dispatch

Al igual que con docopt, con [docopt_dispatch] creas tu `--help` en la variable `__doc__` de tu módulo de punto de entrada. Allí, se llama `dispatch` con la cadena doc como argumento, para que pueda ejecutar el analizador sobre él.

Una vez hecho esto, en lugar de manejar manualmente los argumentos (que por lo general terminan en una estructura ciclomática alta de / else), lo dejas para enviar, dando solo la forma en que quieres manejar el conjunto de argumentos.

Esto es para lo que es el decorador `dispatch.on` : le da el argumento o la secuencia de argumentos que deberían desencadenar la función, y esa función se ejecutará con los valores coincidentes como parámetros.

```
"""Run something in development or production mode.

Usage: run.py --development <host> <port>
       run.py --production <host> <port>
       run.py items add <item>
       run.py items delete <item>

"""
from docopt_dispatch import dispatch

@dispatch.on('--development')
def development(host, port, **kwargs):
    print('in *development* mode')

@dispatch.on('--production')
def development(host, port, **kwargs):
    print('in *production* mode')

@dispatch.on('items', 'add')
def items_add(item, **kwargs):
    print('adding item...')

@dispatch.on('items', 'delete')
def items_delete(item, **kwargs):
    print('deleting item...')

if __name__ == '__main__':
    dispatch(__doc__)
```

Lea Análisis de argumentos de línea de comandos en línea:
https://riptutorial.com/es/python/topic/1382/analisis-de-argumentos-de-linea-de-comandos

Capítulo 12: Análisis de HTML

Examples

Localiza un texto después de un elemento en BeautifulSoup.

Imagina que tienes el siguiente HTML:

```
<div>
    <label>Name:</label>
    John Smith
</div>
```

Y necesitas ubicar el texto "John Smith" después del elemento de `label` .

En este caso, puede ubicar el elemento de `label` por texto y luego usar la propiedad `.next_sibling` :

```
from bs4 import BeautifulSoup

data = """
<div>
    <label>Name:</label>
    John Smith
</div>
"""

soup = BeautifulSoup(data, "html.parser")

label = soup.find("label", text="Name:")
print(label.next_sibling.strip())
```

Imprime `John Smith` .

Usando selectores de CSS en BeautifulSoup

BeautifulSoup tiene un soporte limitado para los selectores de CSS , pero cubre los más utilizados. Use el método `select()` para encontrar múltiples elementos y `select_one()` para encontrar un solo elemento.

Ejemplo básico:

```
from bs4 import BeautifulSoup

data = """
<ul>
    <li class="item">item1</li>
    <li class="item">item2</li>
    <li class="item">item3</li>
</ul>
"""
```

```
soup = BeautifulSoup(data, "html.parser")

for item in soup.select("li.item"):
    print(item.get_text())
```

Huellas dactilares:

```
item1
item2
item3
```

PyQuery

pyquery es una biblioteca tipo jquery para python. Tiene muy buen soporte para selectores css.

```
from pyquery import PyQuery

html = """
<h1>Sales</h1>
<table id="table">
<tr>
    <td>Lorem</td>
    <td>46</td>
</tr>
<tr>
    <td>Ipsum</td>
    <td>12</td>
</tr>
<tr>
    <td>Dolor</td>
    <td>27</td>
</tr>
<tr>
    <td>Sit</td>
    <td>90</td>
</tr>
</table>
"""

doc = PyQuery(html)

title = doc('h1').text()

print title

table_data = []

rows = doc('#table > tr')
for row in rows:
    name = PyQuery(row).find('td').eq(0).text()
    value = PyQuery(row).find('td').eq(1).text()

    print "%s\t  %s" % (name, value)
```

Lea Análisis de HTML en línea: https://riptutorial.com/es/python/topic/1384/analisis-de-html

Capítulo 13: Anti-patrones de Python

Examples

Con exceso de celo excepto la cláusula

Las excepciones son poderosas, pero una sola cláusula exagerada puede quitarlo todo en una sola línea.

```
try:
    res = get_result()
    res = res[0]
    log('got result: %r' % res)
except:
    if not res:
        res = ''
    print('got exception')
```

Este ejemplo demuestra 3 síntomas del antipattern:

1. El tipo de excepción `except` sin excepción (línea 5) detectará incluso las excepciones correctas, incluido `KeyboardInterrupt` . Eso evitará que el programa salga en algunos casos.
2. El bloque de excepción no vuelve a generar el error, lo que significa que no podremos saber si la excepción provino de `get_result` o porque `res` era una lista vacía.
3. Lo peor de todo, si nos preocupaba que el resultado estuviera vacío, hemos causado algo mucho peor. Si `get_result` falla, la `res` permanecerá completamente sin configurar, y la referencia a la `res` en el bloque de excepción, generará `NameError` , `NameError` completamente el error original.

Siempre piensa en el tipo de excepción que estás tratando de manejar. Dale una lectura a la página de excepciones y comprueba qué excepciones básicas existen.

Aquí hay una versión fija del ejemplo anterior:

```
import traceback

try:
    res = get_result()
except Exception:
    log_exception(traceback.format_exc())
    raise
try:
    res = res[0]
except IndexError:
    res = ''

log('got result: %r' % res)
```

Capturamos excepciones más específicas, volviendo a subir cuando sea necesario. Unas líneas más, pero infinitamente más correctas.

Mirando antes de saltar con la función de procesador intensivo

Un programa puede fácilmente perder tiempo llamando a una función intensiva en el procesador varias veces.

Por ejemplo, tome una función que se parece a esto: devuelve un entero si el `value` entrada puede producir uno, de lo contrario, `None` :

```
def intensive_f(value): # int -> Optional[int]
   # complex, and time-consuming code
   if process_has_failed:
       return None
   return integer_output
```

Y podría ser utilizado de la siguiente manera:

```
x = 5
if intensive_f(x) is not None:
    print(intensive_f(x) / 2)
else:
    print(x, "could not be processed")

print(x)
```

Si bien esto funcionará, tiene el problema de llamar `intensive_f` , que duplica el tiempo de ejecución del código. Una mejor solución sería obtener el valor de retorno de la función de antemano.

```
x = 5
result = intensive_f(x)
if result is not None:
    print(result / 2)
else:
    print(x, "could not be processed")
```

Sin embargo, una forma más clara y posiblemente más pirónica es usar excepciones, por ejemplo:

```
x = 5
try:
    print(intensive_f(x) / 2)
except TypeError: # The exception raised if None + 1 is attempted
    print(x, "could not be processed")
```

Aquí no se necesita una variable temporal. Puede ser a menudo preferible utilizar una `assert` declaración, y para coger el `AssertionError` lugar.

Claves del diccionario

Un ejemplo común de dónde se puede encontrar esto es acceder a las claves del diccionario. Por ejemplo comparar:

```
bird_speeds = get_very_long_dictionary()

if "european swallow" in bird_speeds:
    speed = bird_speeds["european swallow"]
else:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

con:

```
bird_speeds = get_very_long_dictionary()

try:
    speed = bird_speeds["european swallow"]
except KeyError:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

El primer ejemplo tiene que mirar el diccionario dos veces, y como este es un diccionario largo, puede llevar mucho tiempo hacerlo cada vez. El segundo solo requiere una búsqueda en el diccionario y, por lo tanto, ahorra mucho tiempo de procesador.

Una alternativa a esto es usar `dict.get(key, default)` *, sin embargo, muchas circunstancias pueden requerir operaciones más complejas en el caso de que la clave no esté presente*

Lea Anti-patrones de Python en línea: https://riptutorial.com/es/python/topic/4700/anti-patrones-de-python

Capítulo 14: Apilar

Introducción

Una pila es un contenedor de objetos que se insertan y eliminan de acuerdo con el principio de último en entrar, primero en salir (LIFO). En las pilas de pushdown solo se permiten dos operaciones: **empujar el elemento en la pila y sacar el elemento de la pila** . Una pila es una estructura de datos de acceso limitado: los **elementos se pueden agregar y eliminar de la pila solo en la parte superior** . Aquí hay una definición estructural de una pila: una pila está vacía o consiste en una parte superior y el resto que es una pila.

Sintaxis

- stack = [] # Crea la pila
- stack.append (objeto) # Agregar objeto a la parte superior de la pila
- stack.pop () -> object # Devuelve el objeto más superior de la pila y también lo elimina
- list [-1] -> object # Mira el objeto más superior sin quitarlo

Observaciones

De Wikipedia :

> En informática, una *pila* es un tipo de datos abstracto que sirve como una colección de elementos, con dos operaciones principales: *push* , que agrega un elemento a la colección, y *pop* , que elimina el elemento agregado más reciente que aún no se eliminó.

Debido a la forma en que se accede a sus elementos, las pilas son también conocidos como *último en entrar, primero en salir (LIFO) apila.*

En Python se pueden usar listas como pilas con `append()` como push y `pop()` como operaciones emergentes. Ambas operaciones se ejecutan en tiempo constante O (1).

La estructura de datos `deque` de Python también se puede utilizar como una pila. En comparación con las listas, los `deque` permiten operaciones de inserción y pop con una complejidad de tiempo constante desde ambos extremos.

Examples

Creación de una clase de pila con un objeto de lista

Usando un objeto de `list` , puede crear una pila genérica completamente funcional con métodos auxiliares, como mirar y comprobar si la pila está vacía. Echa un vistazo a los documentos oficiales de Python para utilizar la `list` como `Stack` aquí .

```
#define a stack class
class Stack:
    def __init__(self):
        self.items = []

    #method to check the stack is empty or not
    def isEmpty(self):
        return self.items == []

    #method for pushing an item
    def push(self, item):
        self.items.append(item)

    #method for popping an item
    def pop(self):
        return self.items.pop()

    #check what item is on top of the stack without removing it
    def peek(self):
        return self.items[-1]

    #method to get the size
    def size(self):
        return len(self.items)

    #to view the entire stack
    def fullStack(self):
        return self.items
```

Un ejemplo de ejecución:

```
stack = Stack()
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
print('Pushing integer 1')
stack.push(1)
print('Pushing string "Told you, I am generic stack!"')
stack.push('Told you, I am generic stack!')
print('Pushing integer 3')
stack.push(3)
print('Current stack:', stack.fullStack())
print('Popped item:', stack.pop())
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
```

Salida:

```
Current stack: []
Stack empty?: True
Pushing integer 1
Pushing string "Told you, I am generic stack!"
Pushing integer 3
Current stack: [1, 'Told you, I am generic stack!', 3]
Popped item: 3
Current stack: [1, 'Told you, I am generic stack!']
Stack empty?: False
```

Paréntesis de paréntesis

Las pilas se utilizan a menudo para el análisis. Una tarea de análisis simple es verificar si una cadena de paréntesis coincide.

Por ejemplo, la cadena `([])` coincide, porque los corchetes exterior e interior forman pares. `()<>)` no coincide, porque el último `)` no tiene pareja. `([)]` tampoco coincide, porque los pares deben estar completamente dentro o fuera de otros pares.

```
def checkParenth(str):
    stack = Stack()
    pushChars, popChars = "<({[", ">)}]"
    for c in str:
        if c in pushChars:
            stack.push(c)
        elif c in popChars:
            if stack.isEmpty():
                return False
            else:
                stackTop = stack.pop()
                # Checks to see whether the opening bracket matches the closing one
                balancingBracket = pushChars[popChars.index(c)]
                if stackTop != balancingBracket:
                    return False
        else:
            return False

    return not stack.isEmpty()
```

Lea Apilar en línea: https://riptutorial.com/es/python/topic/3807/apilar

Capítulo 15: Árbol de sintaxis abstracta

Examples

Analizar funciones en un script de python

Esto analiza un script de Python y, para cada función definida, informa el número de línea donde comenzó la función, donde termina la firma, donde termina la cadena de documentos y donde termina la definición de la función.

```
#!/usr/local/bin/python3

import ast
import sys

""" The data we collect.  Each key is a function name; each value is a dict
with keys: firstline, sigend, docend, and lastline and values of line numbers
where that happens. """
functions = {}

def process(functions):
    """ Handle the function data stored in functions. """
    for funcname,data in functions.items():
        print("function:",funcname)
        print("\tstarts at line:",data['firstline'])
        print("\tsignature ends at line:",data['sigend'])
        if ( data['sigend'] < data['docend'] ):
            print("\tdocstring ends at line:",data['docend'])
        else:
            print("\tno docstring")
        print("\tfunction ends at line:",data['lastline'])
        print()

class FuncLister(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        """ Recursively visit all functions, determining where each function
        starts, where its signature ends, where the docstring ends, and where
        the function ends. """
        functions[node.name] = {'firstline':node.lineno}
        sigend = max(node.lineno,lastline(node.args))
        functions[node.name]['sigend'] = sigend
        docstring = ast.get_docstring(node)
        docstringlength = len(docstring.split('\n')) if docstring else -1
        functions[node.name]['docend'] = sigend+docstringlength
        functions[node.name]['lastline'] = lastline(node)
        self.generic_visit(node)

def lastline(node):
    """ Recursively find the last line of a node """
    return max( [ node.lineno if hasattr(node,'lineno') else -1 , ]
                +[lastline(child) for child in ast.iter_child_nodes(node)] )

def readin(pythonfilename):
    """ Read the file name and store the function data into functions. """
    with open(pythonfilename) as f:
        code = f.read()
```

```
    FuncLister().visit(ast.parse(code))

def analyze(file,process):
    """ Read the file and process the function data. """
    readin(file)
    process(functions)

if __name__ == '__main__':
    if len(sys.argv)>1:
        for file in sys.argv[1:]:
            analyze(file,process)
    else:
        analyze(sys.argv[0],process)
```

Lea Árbol de sintaxis abstracta en línea: https://riptutorial.com/es/python/topic/5370/arbol-de-sintaxis-abstracta

Capítulo 16: Archivos y carpetas I / O

Introducción

Cuando se trata de almacenar, leer o comunicar datos, trabajar con los archivos de un sistema operativo es tanto necesario como fácil con Python. A diferencia de otros idiomas en los que la entrada y salida de archivos requiere objetos complejos de lectura y escritura, Python simplifica el proceso, ya que solo necesita comandos para abrir, leer / escribir y cerrar el archivo. Este tema explica cómo Python puede interactuar con archivos en el sistema operativo.

Sintaxis

- file_object = open (filename [, access_mode] [, buffering])

Parámetros

Parámetro	Detalles
nombre del archivo	la ruta a su archivo o, si el archivo está en el directorio de trabajo, el nombre de archivo de su archivo
modo de acceso	un valor de cadena que determina cómo se abre el archivo
amortiguación	un valor entero utilizado para el búfer de línea opcional

Observaciones

Evitar el infierno de codificación multiplataforma

Cuando se utiliza el `open()` incorporado de Python, es una buena práctica pasar siempre el argumento de `encoding` , si pretende que su código se ejecute en varias plataformas. El motivo de esto es que la codificación predeterminada de un sistema difiere de una plataforma a otra.

Si bien los sistemas `linux` sí usan `utf-8` como predeterminado, esto ***no*** es necesariamente cierto para MAC y Windows.

Para verificar la codificación predeterminada de un sistema, intente esto:

```
import sys
sys.getdefaultencoding()
```

de cualquier intérprete de python.

Por lo tanto, es aconsejable siempre sepcificar una codificación, para asegurarse de que las cadenas con las que está trabajando estén codificadas como lo que cree que son, lo que garantiza la compatibilidad entre plataformas.

```
with open('somefile.txt', 'r', encoding='UTF-8') as f:
    for line in f:
        print(line)
```

Examples

Modos de archivo

Hay diferentes modos con los que puede abrir un archivo, especificados por el parámetro de `mode` . Éstos incluyen:

- `'r'` - modo de lectura. El valor por defecto. Le permite solo leer el archivo, no modificarlo. Al usar este modo el archivo debe existir.
- `'w'` - modo de escritura. Creará un nuevo archivo si no existe, de lo contrario borrará el archivo y le permitirá escribir en él.
- `'a'` - modo de añadir. Escribirá los datos al final del archivo. No borra el archivo, y el archivo debe existir para este modo.
- `'rb'` - modo de lectura en binario. Esto es similar a `r` excepto que la lectura se fuerza en modo binario. Esta es también una opción por defecto.
- `'r+'` - modo de lectura más modo de escritura al mismo tiempo. Esto le permite leer y escribir en archivos al mismo tiempo sin tener que usar `r` y `w` .
- `'rb+'` - modo de lectura y escritura en binario. Lo mismo que `r+` excepto que los datos están en binario
- `'wb'` - modo de escritura en binario. Lo mismo que `w` excepto que los datos están en binario.
- `'w+'` - modo de escritura y lectura. Exactamente igual que `r+` pero si el archivo no existe, se crea uno nuevo. De lo contrario, el archivo se sobrescribe.
- `'wb+'` - modo de escritura y lectura en modo binario. Lo mismo que `w+` pero los datos están en binario.
- `'ab'` - añadiendo en modo binario. Similar a `a` excepto que los datos están en binario.
- `'a+'` - modo de añadir y leer. Similar a `w+` ya que creará un nuevo archivo si el archivo no existe. De lo contrario, el puntero del archivo se encuentra al final del archivo, si existe.
- `'ab+'` - modo de añadir y leer en binario. Lo mismo que `a+` excepto que los datos están en

binario.

```
with open(filename, 'r') as f:
    f.read()
with open(filename, 'w') as f:
    f.write(filedata)
with open(filename, 'a') as f:
    f.write('\n' + newdata)
```

	r	r +	w	w +	una	a +
Leer	✔	✔	✘	✔	✘	✔
Escribir	✘	✔	✔	✔	✔	✔
Crea archivo	✘	✘	✔	✔	✔	✔
Borrar archivo	✘	✘	✔	✔	✘	✘
Posición inicial	comienzo	comienzo	comienzo	comienzo	Fin	Fin

Python 3 agregó un nuevo modo para la `exclusive creation` para que no truncas o sobrescribas accidentalmente un archivo existente.

- `'x'` - abierto para creación exclusiva, generará `FileExistsError` si el archivo ya existe
- `'xb'` - abierto para el modo de escritura de creación exclusiva en binario. Lo mismo que `x` excepto que los datos están en binario.
- `'x+'` - modo de lectura y escritura. Similar a `w+` ya que creará un nuevo archivo si el archivo no existe. De lo contrario, se levantará `FileExistsError` .
- `'xb+'` - modo de escritura y lectura. Exactamente lo mismo que `x+` pero los datos son binarios.

	X	x +
Leer	✘	✔
Escribir	✔	✔
Crea archivo	✔	✔
Borrar archivo	✘	✘
Posición inicial	comienzo	comienzo

Permita que uno escriba su código de archivo abierto de una manera más pitónica:

Python 3.x 3.3

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
    # Your error handling goes here
```

En Python 2 habrías hecho algo como

Python 2.x 2.0

```
import os.path
if os.path.isfile(fname):
    with open("fname", "w") as fout:
        # Work with your open file
else:
    # Your error handling goes here
```

Leyendo un archivo línea por línea

La forma más sencilla de iterar sobre un archivo línea por línea:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

`readline()` permite un control más granular sobre la iteración línea por línea. El siguiente ejemplo es equivalente al de arriba:

```
with open('myfile.txt', 'r') as fp:
    while True:
        cur_line = fp.readline()
        # If the result is an empty string
        if cur_line == '':
            # We have reached the end of the file
            break
        print(cur_line)
```

Usar el iterador de bucle for y readline () juntos se considera una mala práctica.

Más comúnmente, el método `readlines()` se usa para almacenar una colección iterable de las líneas del archivo:

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

Esto imprimiría lo siguiente:

> Línea 0: hola
>
> Línea 1: mundo

Obtener el contenido completo de un archivo

El método preferido para el archivo i / o es usar la palabra clave `with` . Esto asegurará que el identificador de archivo se cierre una vez que se haya completado la lectura o escritura.

```
with open('myfile.txt') as in_file:
    content = in_file.read()

print(content)
```

o, para manejar el cierre del archivo de forma manual, se puede renunciar `with` y simplemente llamar a `close` a sí mismo:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
in_file.close()
```

Tenga en cuenta que sin utilizar una instrucción `with` , puede mantener el archivo abierto por accidente en caso de que surja una excepción inesperada:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close()  # This will never be called
```

Escribiendo en un archivo

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

Si abres `myfile.txt` , verás que su contenido es:

> Línea 1Línea 2Línea 3Línea 4

Python no agrega automáticamente saltos de línea, debe hacerlo manualmente:

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1\n")
    f.write("Line 2\n")
    f.write("Line 3\n")
    f.write("Line 4\n")
```

> Línea 1
> Línea 2
> Línea 3
> Linea 4

No use `os.linesep` como terminador de línea al escribir archivos abiertos en modo de texto (el valor predeterminado); use `\n` lugar.

Si desea especificar una codificación, simplemente agregue el parámetro de `encoding` a la función de `open` :

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

También es posible utilizar la declaración de impresión para escribir en un archivo. La mecánica es diferente en Python 2 vs Python 3, pero el concepto es el mismo en que puedes tomar la salida que habría ido a la pantalla y enviarla a un archivo.

Python 3.x 3.0

```
with open('fred.txt', 'w') as outfile:
    s = "I'm Not Dead Yet!"
    print(s) # writes to stdout
    print(s, file = outfile) # writes to outfile

    #Note: it is possible to specify the file parameter AND write to the screen
    #by making sure file ends up with a None value either directly or via a variable
    myfile = None
    print(s, file = myfile) # writes to stdout
    print(s, file = None)   # writes to stdout
```

En Python 2 habrías hecho algo como

Python 2.x 2.0

```
outfile = open('fred.txt', 'w')
s = "I'm Not Dead Yet!"
print s   # writes to stdout
print >> outfile, s   # writes to outfile
```

A diferencia de usar la función de escritura, la función de impresión agrega automáticamente saltos de línea.

Copiando los contenidos de un archivo a un archivo diferente

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- Usando el módulo de `shutil` :

```
import shutil
shutil.copyfile(src, dst)
```

Compruebe si existe un archivo o ruta

Emplea el estilo de codificación EAFP e `try` abrirlo.

```
import errno

try:
    with open(path) as f:
        # File exists
except IOError as e:
    # Raise the exception if it is not ENOENT (No such file or directory)
    if e.errno != errno.ENOENT:
        raise
    # No such file or directory
```

Esto también evitará condiciones de carrera si otro proceso eliminó el archivo entre la verificación y cuando se utiliza. Esta condición de carrera podría ocurrir en los siguientes casos:

- Usando el módulo `os` :

  ```
  import os
  os.path.isfile('/path/to/some/file.txt')
  ```

Python 3.x 3.4

- Utilizando `pathlib` :

  ```
  import pathlib
  path = pathlib.Path('/path/to/some/file.txt')
  if path.is_file():
      ...
  ```

Para verificar si existe una ruta determinada o no, puede seguir el procedimiento anterior de EAFP o verificar explícitamente la ruta:

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## Do stuff
```

Copiar un árbol de directorios

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

El directorio de destino **no debe existir** ya.

Iterar archivos (recursivamente)

Para iterar todos los archivos, incluidos los subdirectorios, use os.walk:

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

root_dir puede ser "." para comenzar desde el directorio actual, o cualquier otra ruta desde la que comenzar.

Python 3.x 3.5

Si también desea obtener información sobre el archivo, puede usar el método más eficiente os.scandir así:

```
for entry in os.scandir(path):
   if not entry.name.startswith('.') and entry.is_file():
       print(entry.name)
```

Leer un archivo entre un rango de líneas.

Supongamos que desea iterar solo entre algunas líneas específicas de un archivo

Puedes hacer uso de `itertools` para eso.

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

Esto leerá las líneas 13 a 20, ya que la indexación de Python comienza desde 0. Por lo tanto, la línea número 1 se indexa como 0

Como también puede leer algunas líneas adicionales haciendo uso de la `next()` palabra clave `next()` aquí.

Y cuando esté utilizando el objeto de archivo como un iterable, no use la instrucción `readline()` aquí ya que las dos técnicas para atravesar un archivo no deben mezclarse

Acceso aleatorio a archivos usando mmap

El uso del módulo `mmap` permite al usuario acceder aleatoriamente a las ubicaciones de un archivo asignando el archivo a la memoria. Esta es una alternativa al uso de operaciones de archivos normales.

```
import mmap

with open('filename.ext', 'r') as fd:
    # 0: map the whole file
    mm = mmap.mmap(fd.fileno(), 0)

    # print characters at indices 5 through 10
```

```
    print mm[5:10]

    # print the line starting from mm's current position
    print mm.readline()

    # write a character to the 5th index
    mm[5] = 'a'

    # return mm's position to the beginning of the file
    mm.seek(0)

    # close the mmap object
    mm.close()
```

Reemplazo de texto en un archivo

```
import fileinput

replacements = {'Search1': 'Replace1',
                'Search2': 'Replace2'}

for line in fileinput.input('filename.txt', inplace=True):
    for search_for in replacements:
        replace_with = replacements[search_for]
        line = line.replace(search_for, replace_with)
    print(line, end='')
```

Comprobando si un archivo está vacío

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

o

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

Sin embargo, ambos lanzarán una excepción si el archivo no existe. Para evitar tener que atrapar tal error, haga esto:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

que devolverá un valor `bool` .

Lea Archivos y carpetas I / O en línea: https://riptutorial.com/es/python/topic/267/archivos-y-carpetas-i---o

Capítulo 17: ArcPy

Observaciones

Este ejemplo utiliza un cursor de búsqueda del módulo de acceso a datos (da) de ArcPy.

No confunda la sintaxis de arcpy.da.SearchCursor con la arcpy.SearchCursor () anterior y más lenta.

El módulo de acceso a datos (arcpy.da) solo ha estado disponible desde ArcGIS 10.1 para escritorio.

Examples

Impresión del valor de un campo para todas las filas de la clase de entidad en la geodatabase de archivos usando el cursor de búsqueda

Para imprimir un campo de prueba (TestField) desde una clase de entidad de prueba (TestFC) en una geodatabase de archivos de prueba (Test.gdb) ubicada en una carpeta temporal (C: \ Temp):

```
with arcpy.da.SearchCursor(r"C:\Temp\Test.gdb\TestFC",["TestField"]) as cursor:
    for row in cursor:
        print row[0]
```

createDissolvedGDB para crear un archivo gdb en el área de trabajo

```
def createDissolvedGDB(workspace, gdbName):
    gdb_name = workspace + "/" + gdbName + ".gdb"

    if(arcpy.Exists(gdb_name):
        arcpy.Delete_management(gdb_name)
        arcpy.CreateFileGDB_management(workspace, gdbName, "")
    else:
        arcpy.CreateFileGDB_management(workspace, gdbName, "")

    return gdb_name
```

Lea ArcPy en línea: https://riptutorial.com/es/python/topic/4693/arcpy

Capítulo 18: Arrays

Introducción

Las "matrices" en Python no son las matrices en lenguajes de programación convencionales como C y Java, sino que están más cerca de las listas. Una lista puede ser una colección de elementos homogéneos o heterogéneos, y puede contener ints, cadenas u otras listas.

Parámetros

Parámetro	Detalles
`b`	Representa un entero con signo de tamaño 1 byte
`B`	Representa un entero sin signo de tamaño 1 byte
`c`	Representa el carácter de 1 byte de tamaño.
`u`	Representa caracteres unicode de tamaño 2 bytes.
`h`	Representa un entero con signo de tamaño 2 bytes
`H`	Representa un entero sin signo de tamaño 2 bytes
`i`	Representa un entero con signo de tamaño 2 bytes
`I`	Representa un entero sin signo de tamaño 2 bytes
`w`	Representa caracteres unicode de tamaño 4 bytes.
`l`	Representa un entero con signo de tamaño 4 bytes
`L`	Representa un entero sin signo de tamaño 4 bytes
`f`	Representa punto flotante de tamaño 4 bytes.
`d`	Representa punto flotante de tamaño 8 bytes.

Examples

Introducción básica a las matrices

Una matriz es una estructura de datos que almacena valores del mismo tipo de datos. En Python, esta es la principal diferencia entre matrices y listas.

Mientras que las listas de python pueden contener valores correspondientes a diferentes tipos de

datos, las matrices en python solo pueden contener valores correspondientes al mismo tipo de datos. En este tutorial, entenderemos las matrices de Python con algunos ejemplos.

Si eres nuevo en Python, comienza con el artículo de Introducción a Python.

Para usar arrays en lenguaje python, necesita importar el módulo de `array` estándar. Esto se debe a que la matriz no es un tipo de datos fundamental como cadenas, números enteros, etc. Aquí se explica cómo puede importar `array` módulo de `array` en Python:

```
from array import *
```

Una vez que haya importado el módulo de `array` , puede declarar una matriz. Así es como lo haces:

```
arrayIdentifierName = array(typecode, [Initializers])
```

En la declaración anterior, `arrayIdentifierName` es el nombre de la matriz, `typecode` permite a python saber el tipo de matriz y los `Initializers` son los valores con los que se inicializa la matriz.

Los códigos de tipo son los códigos que se utilizan para definir el tipo de valores de matriz o el tipo de matriz. La tabla en la sección de parámetros muestra los valores posibles que puede usar al declarar una matriz y su tipo.

Aquí hay un ejemplo del mundo real de declaración de matriz de python:

```
my_array = array('i',[1,2,3,4])
```

En el ejemplo anterior, el código de código utilizado es `i` . Este código de código representa un entero con signo cuyo tamaño es de 2 bytes.

Aquí hay un ejemplo simple de una matriz que contiene 5 enteros

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

Accede a elementos individuales a través de índices.

Se puede acceder a elementos individuales a través de índices. Las matrices de Python están indexadas a cero. Aquí hay un ejemplo :

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
```

```
print(my_array[2])
# 3
print(my_array[0])
# 1
```

Agregue cualquier valor a la matriz usando el método append ()

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Tenga en cuenta que el valor `6` se agregó a los valores de matriz existentes.

Insertar valor en una matriz usando el método insert ()

Podemos usar el método `insert()` para insertar un valor en cualquier índice de la matriz. Aquí hay un ejemplo :

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

En el ejemplo anterior, el valor 0 se insertó en el índice 0. Tenga en cuenta que el primer argumento es el índice, mientras que el segundo argumento es el valor.

Extiende la matriz de python usando el método extend ()

Una matriz de python se puede ampliar con más de un valor utilizando `extend()` método `extend()` . Aquí hay un ejemplo :

```
my_array = array('i', [1,2,3,4,5])
my_extnd_array = array('i', [7,8,9,10])
my_array.extend(my_extnd_array)
# array('i', [1, 2, 3, 4, 5, 7, 8, 9, 10])
```

Vemos que la matriz my_array se extendió con los valores de `my_extnd_array` .

Agregue elementos de la lista a la matriz usando el método fromlist ()

Aquí hay un ejemplo:

```
my_array = array('i', [1,2,3,4,5])
c=[11,12,13]
my_array.fromlist(c)
# array('i', [1, 2, 3, 4, 5, 11, 12, 13])
```

Entonces vemos que los valores 11,12 y 13 se agregaron de la lista `c` a `my_array` .

Elimine cualquier elemento del arreglo usando el método remove ()

Aquí hay un ejemplo :

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

Vemos que el elemento 4 fue eliminado de la matriz.

Eliminar el último elemento de la matriz utilizando el método pop ()

`pop` elimina el último elemento de la matriz. Aquí hay un ejemplo :

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

Así que vemos que el último elemento (`5`) se extrajo de la matriz.

Obtenga cualquier elemento a través de su índice usando el método index ()

`index()` devuelve el primer índice del valor coincidente. Recuerde que las matrices están indexadas a cero.

```
my_array = array('i', [1,2,3,4,5])
print(my_array.index(5))
# 5
my_array = array('i', [1,2,3,3,5])
print(my_array.index(3))
# 3
```

Tenga en cuenta en ese segundo ejemplo que solo se devolvió un índice, aunque el valor existe dos veces en la matriz

Invertir una matriz de python usando el método reverse ()

El método `reverse()` hace lo que el nombre dice que hará: invierte la matriz. Aquí hay un ejemplo :

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

Obtener información de búfer de matriz a través del método buffer_info ()

Este método le proporciona la dirección de inicio del búfer de matriz en la memoria y la cantidad de elementos en la matriz. Aquí hay un ejemplo:

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

Compruebe el número de apariciones de un elemento utilizando el método count ()

`count()` devolverá el número de veces y el elemento aparecerá en una matriz. En el siguiente ejemplo vemos que el valor `3` aparece dos veces.

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

Convierte una matriz en una cadena usando el método tostring ()

`tostring()` convierte la matriz en una cadena.

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

Convierta la matriz a una lista de python con los mismos elementos utilizando el método tolist ()

Cuando necesite un objeto de `list` Python, puede utilizar el método `tolist()` para convertir su matriz en una lista.

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

Agregue una cadena a la matriz de caracteres utilizando el método fromstring ()

Puede agregar una cadena a una matriz de caracteres usando `fromstring()`

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

Lea Arrays en línea: https://riptutorial.com/es/python/topic/4866/arrays

Capítulo 19: Audio

Examples

Audio con pyglet

```
import pyglet
audio = pyglet.media.load("audio.wav")
audio.play()
```

Para más información, ver pyglet.

Trabajando con archivos WAV

WinSound

- Entorno de Windows

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```

ola

- Soporte mono / estéreo
- No soporta compresión / descompresión

```
import wave
with wave.open("path_to_wav_file.wav", "rb") as wav_file:    # Open WAV file in read-only
mode.
    # Get basic information.
    n_channels = wav_file.getnchannels()      # Number of channels. (1=Mono, 2=Stereo).
    sample_width = wav_file.getsampwidth()    # Sample width in bytes.
    framerate = wav_file.getframerate()       # Frame rate.
    n_frames = wav_file.getnframes()          # Number of frames.
    comp_type = wav_file.getcomptype()        # Compression type (only supports "NONE").
    comp_name = wav_file.getcompname()        # Compression name.

    # Read audio data.
    frames = wav_file.readframes(n_frames)    # Read n_frames new frames.
    assert len(frames) == sample_width * n_frames

# Duplicate to a new WAV file.
with wave.open("path_to_new_wav_file.wav", "wb") as wav_file:    # Open WAV file in write-only
mode.
    # Write audio data.
    params = (n_channels, sample_width, framerate, n_frames, comp_type, comp_name)
    wav_file.setparams(params)
    wav_file.writeframes(frames)
```

Convierte cualquier archivo de sonido con python y ffmpeg

```
from subprocess import check_call

ok = check_call(['ffmpeg','-i','input.mp3','output.wav'])
if ok:
    with open('output.wav', 'rb') as f:
        wav_file = f.read()
```

Nota:

- http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is-there-even-a-difference
- ¿Cuáles son las diferencias y similitudes entre ffmpeg, libav y avconv?

Tocando los pitidos de Windows

Windows proporciona una interfaz explícita a través de la cual el módulo `winsound` permite reproducir sonidos brutos en una frecuencia y duración determinadas.

```
import winsound
freq = 2500 # Set frequency To 2500 Hertz
dur = 1000 # Set duration To 1000 ms == 1 second
winsound.Beep(freq, dur)
```

Lea Audio en línea: https://riptutorial.com/es/python/topic/8189/audio

Capítulo 20: Aumentar errores / excepciones personalizados

Introducción

Python tiene muchas excepciones integradas que obligan a su programa a generar un error cuando algo falla.

Sin embargo, a veces es posible que necesite crear excepciones personalizadas que sirvan a su propósito.

En Python, los usuarios pueden definir dichas excepciones creando una nueva clase. Esta clase de excepción debe derivarse, directa o indirectamente, de la clase de excepción. La mayoría de las excepciones incorporadas también se derivan de esta clase.

Examples

Excepción personalizada

Aquí, hemos creado una excepción definida por el usuario llamada CustomError que se deriva de la clase Exception. Esta nueva excepción se puede generar, como otras excepciones, usando la declaración de aumento con un mensaje de error opcional.

```
class CustomError(Exception):
       pass

x = 1

if x == 1:
    raise CustomError('This is custom error')
```

Salida:

```
Traceback (most recent call last):
  File "error_custom.py", line 8, in <module>
    raise CustomError('This is custom error')
__main__.CustomError: This is custom error
```

Atrapar Excepción personalizada

Este ejemplo muestra cómo capturar Excepciones personalizadas

```
class CustomError(Exception):
     pass

try:
    raise CustomError('Can you catch me ?')
```

```
except CustomError as e:
    print ('Catched CustomError :{}'.format(e))
except Exception as e:
    print ('Generic exception: {}'.format(e))
```

Salida:

```
Catched CustomError :Can you catch me ?
```

Lea Aumentar errores / excepciones personalizados en línea:
https://riptutorial.com/es/python/topic/10882/aumentar-errores---excepciones-personalizados

Capítulo 21: Biblioteca de subproceso

Sintaxis

- subprocess.call (args, *, stdin = None, stdout = None, stderr = None, shell = False, timeout = None)
- subprocess.Popen (args, bufsize = -1, executable = None, stdin = None, stdout = None, stderr = None, preexec_fn = None, close_fds = True, shell = False, cwd = None, env = None, universal_newlines = False , startupinfo = Ninguna, creationflags = 0, restore_signals = True, start_new_session = False, pass_fds = ())

Parámetros

Parámetro	Detalles
`args`	Un solo ejecutable, o secuencia de ejecutables y argumentos - `'ls'` , `['ls', '-la']`
`shell`	Ejecutar bajo una cáscara? El shell predeterminado para `/bin/sh` en POSIX.
`cwd`	Directorio de trabajo del proceso hijo.

Examples

Llamando Comandos Externos

El caso de uso más simple es usar la función `subprocess.call` . Acepta una lista como primer argumento. El primer elemento de la lista debe ser la aplicación externa a la que desea llamar. Los otros elementos de la lista son argumentos que se pasarán a esa aplicación.

```
subprocess.call([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

Para los comandos de shell, establezca `shell=True` y proporcione el comando como una cadena en lugar de una lista.

```
subprocess.call('echo "Hello, world"', shell=True)
```

Tenga en cuenta que los dos comandos anteriores solo devuelven el `exit status` de `exit status` del subproceso. Además, preste atención cuando use `shell=True` ya que proporciona problemas de seguridad (consulte aquí).

Si desea poder obtener la salida estándar del subproceso, sustituya el `subprocess.call` con `subprocess.check_output` . Para un uso más avanzado, refiérase a esto .

Más flexibilidad con Popen

El uso de `subprocess.Popen` proporciona un control más preciso sobre los procesos iniciados que `subprocess.call` .

Lanzar un subproceso

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

La firma para `Popen` es muy similar a la función de `call` ; sin embargo, `Popen` regresará de inmediato en lugar de esperar a que se complete el subproceso como lo hace la `call` .

Esperando en un subproceso para completar

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
process.wait()
```

Salida de lectura de un subproceso

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)

# This will block until process completes
stdout, stderr = process.communicate()
print stdout
print stderr
```

Acceso interactivo a subprocesos en ejecución.

Puede leer y escribir en `stdin` y `stdout` incluso cuando el subproceso no se haya completado. Esto podría ser útil al automatizar la funcionalidad en otro programa.

Escribiendo a un subproceso

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout = subprocess.PIPE, stdin =
subprocess.PIPE)


process.stdin.write('line of input\n') # Write input

line  = process.stdout.readline() # Read a line from stdout
```

```
# Do logic on line read.
```

Sin embargo, si solo necesita un conjunto de entrada y salida, en lugar de una interacción dinámica, debe usar `communicate()` lugar de acceder directamente a `stdin` y `stdout` .

Leyendo un stream desde un subproceso

En caso de que quiera ver la salida de un subproceso línea por línea, puede usar el siguiente fragmento de código:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

en el caso de que la salida del subcomando no tenga un carácter EOL, el fragmento de código anterior no funciona. A continuación, puede leer el carácter de salida por carácter de la siguiente manera:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

El `1` especificado como argumento a la `read` método se indica a leer a leer 1 carácter cada vez. Puede especificar leer tantos caracteres como desee utilizando un número diferente. El número negativo o 0 indica que se `read` para leer como una sola cadena hasta que se encuentre el EOF (consulte aquí).

En los dos fragmentos de código anteriores, `process.poll()` es `None` hasta que finalice el subproceso. Esto se usa para salir del bucle una vez que no hay más salida para leer.

El mismo procedimiento podría aplicarse al `stderr` del subproceso.

Cómo crear el argumento de la lista de comandos

El método de subproceso que permite ejecutar comandos necesita el comando en forma de una lista (al menos usando `shell_mode=True`).

Las reglas para crear la lista no siempre son sencillas de seguir, especialmente con comandos complejos. Afortunadamente, hay una herramienta muy útil que permite hacer eso: `shlex` . La forma más fácil de crear la lista que se utilizará como comando es la siguiente:

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

Un ejemplo simple:

```
import shlex
```

```
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

Lea Biblioteca de subproceso en línea: https://riptutorial.com/es/python/topic/1393/biblioteca-de-subproceso

Capítulo 22: Bloques de código, marcos de ejecución y espacios de nombres.

Introducción

Un bloque de código es una parte del texto del programa Python que se puede ejecutar como una unidad, como un módulo, una definición de clase o un cuerpo de función. Algunos bloques de código (como módulos) normalmente se ejecutan solo una vez, otros (como cuerpos de función) pueden ejecutarse muchas veces. Los bloques de código pueden contener textualmente otros bloques de código. Los bloques de código pueden invocar otros bloques de código (que pueden o no estar contenidos textualmente en ellos) como parte de su ejecución, por ejemplo, invocando (llamando) una función.

Examples

Espacios de nombres de bloque de código

Tipo de bloque de código	Espacio de nombres global	Espacio de nombres local
Módulo	ns para el modulo	igual que global
Script (archivo o comando)	ns para `__main__`	igual que global
Comando interactivo	ns para `__main__`	igual que global
Definición de clase	ns globales del bloque que contiene	nuevo espacio de nombres
Cuerpo de funcion	ns globales del bloque que contiene	nuevo espacio de nombres
Cadena pasada a la sentencia `exec`	ns globales del bloque que contiene	espacio de nombres local del bloque que contiene
Cadena pasada a `eval()`	ns global de la persona que llama	ns locales de la persona que llama
Archivo leído por `execfile()`	ns global de la persona que llama	ns locales de la persona que llama
Expresión leída por `input()`	ns global de la persona que llama	ns locales de la persona que llama

Lea Bloques de código, marcos de ejecución y espacios de nombres. en línea: https://riptutorial.com/es/python/topic/10741/bloques-de-codigo--marcos-de-ejecucion-y-espacios-de-nombres-

Capítulo 23: Bucles

Introducción

Como una de las funciones más básicas de la programación, los bucles son una pieza importante para casi todos los lenguajes de programación. Los bucles permiten a los desarrolladores configurar ciertas partes de su código para que se repitan a través de una serie de bucles que se conocen como iteraciones. Este tema cubre el uso de múltiples tipos de bucles y aplicaciones de bucles en Python.

Sintaxis

- mientras que <expresión booleana>:
- para <variable> en <iterable>:
- para <variable> en rango (<número>):
- para <variable> en el rango (<start_number>, <end_number>):
- para <variable> en el rango (<start_number>, <end_number>, <step_size>):
- para i, <variable> in enumerate (<iterable>): # con índice i
- para <variable1>, <variable2> en zip (<iterable1>, <iterable2>):

Parámetros

Parámetro	Detalles
expresión booleana	Expresión que se puede evaluar en un contexto booleano, por ejemplo, `x < 10`
variable	Nombre de variable para el elemento actual del `iterable`
iterable	cualquier cosa que implemente iteraciones

Examples

Iterando sobre listas

Para recorrer una lista puedes usar `for` :

```
for x in ['one', 'two', 'three', 'four']:
    print(x)
```

Esto imprimirá los elementos de la lista:

```
one
two
```

```
three
four
```

La función de `range` genera números que también se utilizan a menudo en un bucle for.

```
for x in range(1, 6):
    print(x)
```

El resultado será un tipo de secuencia de rango especial en python> = 3 y una lista en python <= 2. Ambos se pueden pasar usando el bucle for.

```
1
2
3
4
5
```

Si desea recorrer los elementos de una lista *y también* tener un índice para los elementos, puede usar la función de `enumerate` de Python:

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, '::', item)
```

`enumerate` generará tuplas, que se desempaquetan en `index` (un entero) y `item` (el valor real de la lista). El bucle de arriba se imprimirá

```
(0, '::', 'one')
(1, '::', 'two')
(2, '::', 'three')
(3, '::', 'four')
```

Iterar sobre una lista con manipulación de valores usando `map` y `lambda` , es decir, aplicar la función lambda en cada elemento de la lista:

```
x = map(lambda e :  e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

Salida:

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

NB: en Python 3.x `map` devuelve un iterador en lugar de una lista, por lo que, en caso de que necesite una lista, debe emitir la `print(list(x))` del resultado `print(list(x))` (consulte http://www.riptutorial.com/python/ example / 8186 / map-- en http://www.riptutorial.com/python/topic/809/incompatibilities-moving-from-python-2-to-python-3).

Para bucles

`for` bucles, repita una colección de elementos, como `list` o `dict` , y ejecute un bloque de código

con cada elemento de la colección.

```
for i in [0, 1, 2, 3, 4]:
    print(i)
```

Lo anterior `for` bucle se repite sobre una lista de números.

Cada iteración establece el valor de `i` en el siguiente elemento de la lista. Entonces primero será `0` , luego `1` , luego `2` , etc. La salida será la siguiente:

```
0
1
2
3
4
```

`range` es una función que devuelve una serie de números bajo una forma iterable, por lo que se puede utilizar en `for` bucles:

```
for i in range(5):
    print(i)
```

da exactamente el mismo resultado que el primer bucle `for` . Tenga en cuenta que `5` no se imprime, ya que el rango aquí corresponde a los primeros cinco números que cuentan desde `0` .

Objetos iterables e iteradores.

`for` loop puede iterar en cualquier objeto iterable que sea un objeto que defina una función `__getitem__` o `__iter__` . La función `__iter__` devuelve un iterador, que es un objeto con una función `next` que se utiliza para acceder al siguiente elemento de la iterable.

Romper y continuar en bucles

declaración de `break`

Cuando una instrucción de `break` ejecuta dentro de un bucle, el flujo de control se "interrumpe" inmediatamente:

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

El condicional de bucle no se evaluará después de que se ejecute la instrucción `break` . Tenga en

cuenta que las declaraciones de `break` solo se permiten *dentro de los bucles* , sintácticamente. No se puede usar una declaración de `break` dentro de una función para terminar los bucles que llamaron a esa función.

La ejecución de lo siguiente imprime cada dígito hasta el número `4` cuando se cumple la instrucción `break` y el bucle se detiene:

```
0
1
2
3
4
Breaking from loop
```

`break` declaraciones de `break` también se pueden usar dentro `for` bucles, la otra construcción de bucle proporcionada por Python:

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

La ejecución de este bucle ahora imprime:

```
0
1
2
```

Tenga en cuenta que 3 y 4 no se imprimen ya que el bucle ha finalizado.

Si un bucle tiene una cláusula `else` , no se ejecuta cuando el bucle termina a través de una instrucción de `break` .

`continue` declaración

Una instrucción de `continue` pasará a la siguiente iteración del bucle, omitiendo el resto del bloque actual pero continuando el bucle. Al igual que con `break` , `continue` solo puede aparecer dentro de los bucles:

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)

0
1
3
5
```

Tenga en cuenta que `2` y `4` no se imprimen, esto se debe a que `continue` va a la siguiente iteración

en lugar de continuar `print(i)` cuando `i == 2` o `i == 4` .

Bucles anidados

`break` y `continue` solo funciona en un solo nivel de bucle. El siguiente ejemplo solo saldrá del bucle `for` interno, no del bucle `while` externo:

```
while True:
    for i in range(1,5):
        if i == 2:
            break    # Will only break out of the inner loop!
```

Python no tiene la capacidad de romper múltiples niveles de bucles a la vez; si se desea este comportamiento, refactorizar uno o más bucles en una función y reemplazar la `break` con el `return` puede ser la forma de hacerlo.

Usa el `return` desde dentro de una función como un `break`

La declaración de `return` sale de una función, sin ejecutar el código que viene después de ella.

Si tiene un bucle dentro de una función, usar el `return` desde dentro de ese bucle es equivalente a tener una `break` ya que el resto del código del bucle no se ejecuta (*tenga en cuenta que cualquier código después del bucle tampoco se ejecuta*):

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

Si tiene bucles anidados, la instrucción de `return` romperá todos los bucles:

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

saldrá:

```
1 # 1*1
2 # 1*2
3 # 1*3
4 # 1*4
2 # 2*1
4 # 2*2
# return because 2*3 = 6, the remaining iterations of both loops are not executed
```

Bucles con una cláusula "else"

Las declaraciones `for` y `while` (loops) pueden tener opcionalmente una cláusula `else` (en la práctica, este uso es bastante raro).

El `else` cláusula sólo se ejecuta después de un `for` bucle termina por iteración a la terminación, o después de un `while` bucle termina por su expresión condicional convertirse en falsa.

```
for i in range(3):
    print(i)
else:
    print('done')

i = 0
while i < 3:
    print(i)
    i += 1
else:
    print('done')
```

salida:

```
 0
 1
 2
 done
```

La cláusula `else` no *se* ejecuta si el bucle termina de alguna otra manera (a través de una declaración de `break` o al generar una excepción):

```
for i in range(2):
    print(i)
    if i == 1:
        break
else:
    print('done')
```

salida:

```
0
1
```

La mayoría de los otros lenguajes de programación carecen de esta cláusula `else` opcional de bucles. El uso de la palabra clave `else` en particular a menudo se considera confuso.

El concepto original para dicha cláusula se remonta a Donald Knuth y el significado de la palabra clave `else` queda claro si reescribimos un bucle en términos de declaraciones `if` y `goto` de días anteriores a la programación estructurada o de un lenguaje ensamblador de nivel inferior.

Por ejemplo:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
```

es equivalente a:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>

<<end>>:
```

Estos siguen siendo equivalentes si adjuntamos una cláusula `else` a cada uno de ellos.

Por ejemplo:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```

es equivalente a:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

Un bucle `for` con una cláusula `else` puede entenderse de la misma manera. Conceptualmente, hay una condición de bucle que permanece verdadera mientras el objeto o la secuencia iterable aún tenga algunos elementos restantes.

¿Por qué uno usaría esta construcción extraña?

El caso de uso principal para la construcción `for...else` es una implementación concisa de búsqueda como por ejemplo:

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

Para hacer que la `else` de este constructo sea menos confuso, uno puede pensar en él como " *si no se rompe* " o " *si no se encuentra* ".

Algunas discusiones sobre esto se pueden encontrar en [Python-ideas] Resumen de los hilos de for ... else , ¿Por qué python usa 'else' después de los bucles for y while? , y otras cláusulas en declaraciones de bucle

Iterando sobre los diccionarios

Teniendo en cuenta el siguiente diccionario:

```
 d = {"a": 1, "b": 2, "c": 3}
```

Para iterar a través de sus claves, puede utilizar:

```
for key in d:
    print(key)
```

Salida:

```
"a"
"b"
"c"
```

Esto es equivalente a:

```
for key in d.keys():
    print(key)
```

o en Python 2:

```
for key in d.iterkeys():
    print(key)
```

Para iterar a través de sus valores, use:

```
for value in d.values():
```

```
    print(value)
```

Salida:

```
1
2
3
```

Para iterar a través de sus claves y valores, use:

```
for key, value in d.items():
    print(key, "::", value)
```

Salida:

```
a :: 1
b :: 2
c :: 3
```

Tenga en cuenta que en Python 2, `.keys()` , `.values()` y `.items()` devuelven un objeto de `list` . Si simplemente necesita iterar el resultado, puede usar el `.iterkeys()` , `.itervalues()` y `.iteritems()` .

La diferencia entre `.keys()` y `.iterkeys()` , `.values()` y `.itervalues()` , `.items()` y `.iteritems()` es que los métodos `iter*` son generadores. Por lo tanto, los elementos dentro del diccionario se producen uno por uno a medida que se evalúan. Cuando se devuelve un objeto de `list` , todos los elementos se empaquetan en una lista y luego se devuelven para una evaluación adicional.

> Tenga en cuenta también que en Python 3, el orden de los elementos impresos de la manera anterior no sigue ningún orden.

Mientras bucle

A `while` bucle hará que las sentencias de bucle que se ejecutarán hasta que la condición de bucle es Falsey . El siguiente código ejecutará las instrucciones de bucle un total de 4 veces.

```
i = 0
while i < 4:
    #loop statements
    i = i + 1
```

Mientras que el bucle de arriba fácilmente se puede traducir en una más elegante `for` bucle, `while` los bucles son útiles para comprobar si alguna condición se ha cumplido. El siguiente bucle continuará ejecutándose hasta que `myObject` esté listo.

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

`while` bucles también pueden ejecutarse sin una condición usando números (complejos o reales) o `True` :

```
import cmath

complex_num = cmath.sqrt(-1)
while complex_num:      # You can also replace complex_num with any number, True or a value of
any type
    print(complex_num)   # Prints 1j forever
```

Si la condición es siempre cierta, el bucle while se ejecutará para siempre (bucle infinito) si no se termina con una instrucción break o return o una excepción.

```
while True:
    print "Infinite loop"
# Infinite loop
# Infinite loop
# Infinite loop
# ...
```

La Declaración de Pase

`pass` es una declaración nulo para cuando una instrucción se requiere por la sintaxis Python (tal como dentro del cuerpo de un `for` o `while` bucle), pero se requiere ninguna acción o se desee por el programador. Esto puede ser útil como un marcador de posición para el código que aún no se ha escrito.

```
for x in range(10):
    pass #we don't want to do anything, or are not ready to do anything here, so we'll pass
```

En este ejemplo, nada pasará. El bucle `for` se completará sin error, pero no se ejecutará ningún comando o código. `pass` nos permite ejecutar nuestro código con éxito sin tener todos los comandos y acciones completamente implementados.

Del mismo modo, `pass` puede ser utilizado en `while` bucles, así como en las selecciones y definiciones de función, etc.

```
while x == y:
    pass
```

Iterando diferentes partes de una lista con diferentes tamaños de paso

Supongamos que tiene una larga lista de elementos y solo está interesado en todos los demás elementos de la lista. Quizás solo desee examinar los primeros o últimos elementos, o un rango específico de entradas en su lista. Python tiene capacidades integradas de indexación fuertes. Aquí hay algunos ejemplos de cómo lograr estos escenarios.

Aquí hay una lista simple que se utilizará a lo largo de los ejemplos:

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

Iteración sobre toda la lista.

Para iterar sobre cada elemento de la lista, se puede usar un bucle `for` como abajo:

```
for s in lst:
    print s[:1] # print the first letter
```

El bucle `for` asigna s para cada elemento de `lst` . Esto imprimirá:

```
a
b
c
d
e
```

A menudo necesitas tanto el elemento como el índice de ese elemento. La palabra clave `enumerate` realiza esa tarea.

```
for idx, s in enumerate(lst):
    print("%s has an index of %d" % (s, idx))
```

El índice `idx` comenzará con cero e incrementará para cada iteración, mientras que la `s` contendrá el elemento que se está procesando. El fragmento anterior se mostrará:

```
alpha has an index of 0
bravo has an index of 1
charlie has an index of 2
delta has an index of 3
echo has an index of 4
```

Iterar sobre la sub-lista

Si queremos iterar sobre un rango (recordando que Python usa indexación de base cero), use la palabra clave de `range` .

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

Esto daría como resultado:

```
lst at 2 contains charlie
lst at 3 contains delta
```

La lista también puede ser cortada. La siguiente notación de segmento va desde el elemento en

el índice 1 hasta el final con un paso de 2. Los dos bucles `for` dan el mismo resultado.

```
for s in lst[1::2]:
    print(s)

for i in range(1, len(lst), 2):
    print(lst[i])
```

Las salidas de fragmento de código anteriores:

```
bravo
delta
```

La indexación y el corte es un tema propio.

El "half loop" do-while

A diferencia de otros idiomas, Python no tiene una construcción do-until o do-while (esto permitirá que el código se ejecute una vez antes de que se pruebe la condición). Sin embargo, puede combinar un `while True` con una `break` para lograr el mismo propósito.

```
a = 10
while True:
    a = a-1
    print(a)
    if a<7:
        break
print('Done.')
```

Esto imprimirá:

```
9
8
7
6
Done.
```

Bucle y desembalaje

Si quieres recorrer una lista de tuplas por ejemplo:

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

En lugar de hacer algo como esto:

```
for item in collection:
    i1 = item[0]
    i2 = item[1]
    i3 = item[2]
    # logic
```

o algo como esto:

```
for item in collection:
    i1, i2, i3 = item
    # logic
```

Simplemente puede hacer esto:

```
for i1, i2, i3 in collection:
    # logic
```

Esto también funcionará para la *mayoría de los* tipos de iterables, no solo para tuplas.

Lea Bucles en línea: https://riptutorial.com/es/python/topic/237/bucles

Capítulo 24: buscando

Observaciones

Todos los algoritmos de búsqueda en iterables que contienen `n` elementos tienen complejidad `O(n)` . Solo los algoritmos especializados como `bisect.bisect_left()` pueden ser más rápidos con complejidad `O(log(n))` .

Examples

Obtención del índice para cadenas: str.index (), str.rindex () y str.find (), str.rfind ()

`String` también tiene un método de `index` , pero también opciones más avanzadas y la función `str.find` adicional. Para ambos hay un método *inverso* complementario.

```
astring = 'Hello on StackOverflow'
astring.index('o')  # 4
astring.rindex('o') # 20

astring.find('o')   # 4
astring.rfind('o')  # 20
```

La diferencia entre `index` / `rindex` y `find` / `rfind` es lo que sucede si la subcadena no se encuentra en la cadena:

```
astring.index('q') # ValueError: substring not found
astring.find('q')  # -1
```

Todos estos métodos permiten un índice de inicio y finalización:

```
astring.index('o', 5)    # 6
astring.index('o', 6)    # 6 - start is inclusive
astring.index('o', 5, 7) # 6
astring.index('o', 5, 6) #  - end is not inclusive
```

ValueError: subcadena no encontrada

```
astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - still from left to right

astring.rindex('o', 4, 7) # 6
```

Buscando un elemento

Todas las colecciones integradas en Python implementan una forma de verificar la pertenencia al elemento usando `in` .

Lista

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist   # True
10 in alist  # False
```

Tupla

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple    # False
'4' in atuple  # True
```

Cuerda

```
astring = 'i am a string'
'a' in astring   # True
'am' in astring  # True
'I' in astring   # False
```

Conjunto

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset  # True
10 in aset        # False
```

Dictado

`dict` es un poco especial: lo normal `in` sólo comprueba las *teclas.* Si desea buscar en *valores* necesita especificarlo. Lo mismo si quieres buscar pares *clave-valor* .

```
adict = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
1 in adict                 # True   - implicitly searches in keys
'a' in adict               # False
2 in adict.keys()          # True   - explicitly searches in keys
'a' in adict.values()      # True   - explicitly searches in values
(0, 'a') in adict.items()  # True   - explicitly searches key/value pairs
```

Obtención de la lista de índice y las tuplas: list.index (), tuple.index ()

`list` y `tuple` tienen un método de `index` para obtener la posición del elemento:

```
alist = [10, 16, 26, 5, 2, 19, 105, 26]
# search for 16 in the list
alist.index(16) # 1
alist[1]        # 16

alist.index(15)
```

ValueError: 15 no está en la lista

Pero solo devuelve la posición del primer elemento encontrado:

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26)   # 2
atuple[2]          # 26
atuple[7]          # 26 - is also 26!
```

Buscando clave (s) para un valor en dict

`dict` no tiene un método incorporado para buscar un valor o clave porque los *diccionarios* no están ordenados. Puede crear una función que obtenga la clave (o claves) para un valor específico:

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[key] == value:
            foundkeys.append(key)
    return foundkeys
```

Esto también podría escribirse como una lista de comprensión equivalente:

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

Si solo te importa una clave encontrada:

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

Las primeras dos funciones devolverán una `list` de todas las `keys` que tienen el valor especificado:

```
adict = {'a': 10, 'b': 20, 'c': 10}
getKeysForValue(adict, 10)     # ['c', 'a'] - order is random could as well be ['a', 'c']
getKeysForValueComp(adict, 10) # ['c', 'a'] - dito
getKeysForValueComp(adict, 20) # ['b']
getKeysForValueComp(adict, 25) # []
```

El otro solo devolverá una clave:

```
getOneKeyForValue(adict, 10)   # 'c'  - depending on the circumstances this could also be 'a'
getOneKeyForValue(adict, 20)   # 'b'
```

y levante un `StopIteration` - `Exception` si el valor no está en el `dict` :

```
getOneKeyForValue(adict, 25)
```

    StopIteration

Obtención del índice para secuencias ordenadas: bisect.bisect_left ()

Las secuencias ordenadas permiten el uso de algoritmos de búsqueda más rápidos: `bisect.bisect_left()` [1] :

```
import bisect

def index_sorted(sorted_seq, value):
    """Locate the leftmost value exactly equal to x or raise a ValueError"""
    i = bisect.bisect_left(sorted_seq, value)
    if i != len(sorted_seq) and sorted_seq[i] == value:
        return i
    raise ValueError

alist = [i for i in range(1, 100000, 3)] # Sorted list from 1 to 100000 with step 3
index_sorted(alist, 97285) # 32428
index_sorted(alist, 4)     # 1
index_sorted(alist, 97286)
```

ValueError

Para **secuencias clasificadas** muy grandes **,** la ganancia de velocidad puede ser bastante alta. En caso de que la primera búsqueda sea aproximadamente 500 veces más rápida:

```
%timeit index_sorted(alist, 97285)
# 100000 loops, best of 3: 3 µs per loop
%timeit alist.index(97285)
# 1000 loops, best of 3: 1.58 ms per loop
```

Si bien es un poco más lento si el elemento es uno de los primeros:

```
%timeit index_sorted(alist, 4)
# 100000 loops, best of 3: 2.98 µs per loop
%timeit alist.index(4)
# 1000000 loops, best of 3: 580 ns per loop
```

Buscando secuencias anidadas

La búsqueda en secuencias anidadas como una `list` de `tuple` requiere un enfoque como la búsqueda de valores en `dict` pero necesita funciones personalizadas.

El índice de la secuencia más externa si el valor se encontró en la secuencia:

```
def outer_index(nested_sequence, value):
    return next(index for index, inner in enumerate(nested_sequence)
                      for item in inner
                      if item == value)

alist_of_tuples = [(4, 5, 6), (3, 1, 'a'), (7, 0, 4.3)]
outer_index(alist_of_tuples, 'a')  # 1
outer_index(alist_of_tuples, 4.3)  # 2
```

o el índice de la secuencia externa e interna:

```
def outer_inner_index(nested_sequence, value):
    return next((oindex, iindex) for oindex, inner in enumerate(nested_sequence)
                                 for iindex, item in enumerate(inner)
                                 if item == value)

outer_inner_index(alist_of_tuples, 'a') # (1, 2)
alist_of_tuples[1][2]  # 'a'

outer_inner_index(alist_of_tuples, 7)   # (2, 0)
alist_of_tuples[2][0]  # 7
```

En general (*no siempre*) el uso del `next` y una **expresión generadora** con condiciones para encontrar la primera aparición del valor buscado es el enfoque más eficiente.

Búsqueda en clases personalizadas: __contains__ y __iter__

Para permitir el uso de `in` para clases personalizadas, la clase debe proporcionar el método mágico `__contains__` o, en su defecto, un método `__iter__` .

Supongamos que tiene una clase que contiene una `list` de `list` s:

```
class ListList:
    def __init__(self, value):
        self.value = value
        # Create a set of all values for fast access
        self.setofvalues = set(item for sublist in self.value for item in sublist)

    def __iter__(self):
        print('Using __iter__.')
        # A generator over all sublist elements
        return (item for sublist in self.value for item in sublist)

    def __contains__(self, value):
        print('Using __contains__.')
        # Just lookup if the value is in the set
        return value in self.setofvalues

        # Even without the set you could use the iter method for the contains-check:
        # return any(item == value for item in iter(self))
```

Usar la prueba de membresía es posible usando `in` :

```
a = ListList([[1,1,1],[0,1,1],[1,5,1]])
10 in a    # False
# Prints: Using __contains__.
5 in a     # True
# Prints: Using __contains__.
```

Incluso después de eliminar el método `__contains__` :

```
del ListList.__contains__
5 in a     # True
# Prints: Using __iter__.
```

Nota: El bucle `in` (como en `for i in a`) siempre usará `__iter__` incluso si la clase implementa un método `__contains__` .

Lea buscando en línea: https://riptutorial.com/es/python/topic/350/buscando

Capítulo 25: Características ocultas

Examples

Sobrecarga del operador

Todo en Python es un objeto. Cada objeto tiene algunos métodos internos especiales que utiliza para interactuar con otros objetos. En general, estos métodos siguen la convención de denominación `__action__` . En conjunto, esto se denomina como el modelo de datos de Python .

Puede sobrecargar *cualquiera* de estos métodos. Esto se usa comúnmente en la sobrecarga de operadores en Python. A continuación se muestra un ejemplo de sobrecarga de operadores utilizando el modelo de datos de Python. La clase `Vector` crea un vector simple de dos variables. Agregaremos el soporte adecuado para operaciones matemáticas de dos vectores utilizando la sobrecarga de operadores.

```
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, v):
        # Addition with another vector.
        return Vector(self.x + v.x, self.y + v.y)

    def __sub__(self, v):
        # Subtraction with another vector.
        return Vector(self.x - v.x, self.y - v.y)

    def __mul__(self, s):
        # Multiplication with a scalar.
        return Vector(self.x * s, self.y * s)

    def __div__(self, s):
        # Division with a scalar.
        float_s = float(s)
        return Vector(self.x / float_s, self.y / float_s)

    def __floordiv__(self, s):
        # Division with a scalar (value floored).
        return Vector(self.x // s, self.y // s)

    def __repr__(self):
        # Print friendly representation of Vector class. Else, it would
        # show up like, <__main__.Vector instance at 0x01DDDDC8>.
        return '<Vector (%f, %f)>' % (self.x, self.y, )

a = Vector(3, 5)
b = Vector(2, 7)

print a + b # Output: <Vector (5.000000, 12.000000)>
print b - a # Output: <Vector (-1.000000, 2.000000)>
print b * 1.3 # Output: <Vector (2.600000, 9.100000)>
print a // 17 # Output: <Vector (0.000000, 0.000000)>
```

```
print a / 17 # Output: <Vector (0.176471, 0.294118)>
```

El ejemplo anterior demuestra la sobrecarga de operadores numéricos básicos. Una lista completa se puede encontrar aquí .

Lea Características ocultas en línea: https://riptutorial.com/es/python/topic/946/caracteristicas-ocultas

Capítulo 26: ChemPy - paquete de python

Introducción

ChemPy es un paquete de Python diseñado principalmente para resolver y abordar problemas en Química física, analítica e inorgánica. Es un conjunto de herramientas gratuito de código abierto de Python para aplicaciones de química, ingeniería química y ciencia de materiales.

Examples

Fórmulas de análisis

```
from chempy import Substance
ferricyanide = Substance.from_formula('Fe(CN)6-3')
ferricyanide.composition == {0: -3, 26: 1, 6: 6, 7: 6}
True
print(ferricyanide.unicode_name)
Fe(CN)₆³⁻
 print(ferricyanide.latex_name + ", " + ferricyanide.html_name)
Fe(CN)_{6}^{3-}, Fe(CN)<sub>6</sub><sup>3-</sup>
 print('%.3f' % ferricyanide.mass)
211.955
```

En composición, los números atómicos (y 0 para carga) se usan como claves y el recuento de cada tipo se convirtió en valor respectivo.

Equilibrio de la estequiometría de una reacción química.

```
 from chempy import balance_stoichiometry  # Main reaction in NASA's booster rockets:
 reac, prod = balance_stoichiometry({'NH4ClO4', 'Al'}, {'Al2O3', 'HCl', 'H2O', 'N2'})
 from pprint import pprint
 pprint(reac)
{'Al': 10, 'NH4ClO4': 6}
 pprint(prod)
{'Al2O3': 5, 'H2O': 9, 'HCl': 6, 'N2': 3}
 from chempy import mass_fractions
 for fractions in map(mass_fractions, [reac, prod]):
...     pprint({k: '{0:.3g} wt%'.format(v*100) for k, v in fractions.items()})
...
{'Al': '27.7 wt%', 'NH4ClO4': '72.3 wt%'}
{'Al2O3': '52.3 wt%', 'H2O': '16.6 wt%', 'HCl': '22.4 wt%', 'N2': '8.62 wt%'}
```

Reacciones de equilibrio

```
from chempy import Equilibrium
 from sympy import symbols
 K1, K2, Kw = symbols('K1 K2 Kw')
 e1 = Equilibrium({'MnO4-': 1, 'H+': 8, 'e-': 5}, {'Mn+2': 1, 'H2O': 4}, K1)
 e2 = Equilibrium({'O2': 1, 'H2O': 2, 'e-': 4}, {'OH-': 4}, K2)
 coeff = Equilibrium.eliminate([e1, e2], 'e-')
```

```
 coeff
[4, -5]
 redox = e1*coeff[0] + e2*coeff[1]
 print(redox)
20 OH- + 32 H+ + 4 MnO4- = 26 H2O + 4 Mn+2 + 5 O2; K1**4/K2**5
 autoprot = Equilibrium({'H2O': 1}, {'H+': 1, 'OH-': 1}, Kw)
 n = redox.cancel(autoprot)
 n
20
 redox2 = redox + n*autoprot
 print(redox2)
12 H+ + 4 MnO4- = 4 Mn+2 + 5 O2 + 6 H2O; K1**4*Kw**20/K2**5
```

Equilibrios quimicos

```
 from chempy import Equilibrium
 from chempy.chemistry import Species
 water_autop = Equilibrium({'H2O'}, {'H+', 'OH-'}, 10**-14)  # unit "molar" assumed
 ammonia_prot = Equilibrium({'NH4+'}, {'NH3', 'H+'}, 10**-9.24)  # same here
 from chempy.equilibria import EqSystem
 substances = map(Species.from_formula, 'H2O OH- H+ NH3 NH4+'.split())
 eqsys = EqSystem([water_autop, ammonia_prot], substances)
 print('\n'.join(map(str, eqsys.rxns)))  # "rxns" short for "reactions"
H2O = H+ + OH-; 1e-14
NH4+ = H+ + NH3; 5.75e-10
 from collections import defaultdict
 init_conc = defaultdict(float, {'H2O': 1, 'NH3': 0.1})
 x, sol, sane = eqsys.root(init_conc)
 assert sol['success'] and sane
 print(sorted(sol.keys()))  # see package "pyneqsys" for more info
['fun', 'intermediate_info', 'internal_x_vecs', 'nfev', 'njev', 'success', 'x', 'x_vecs']
 print(', '.join('%.2g' % v for v in x))
1, 0.0013, 7.6e-12, 0.099, 0.0013
```

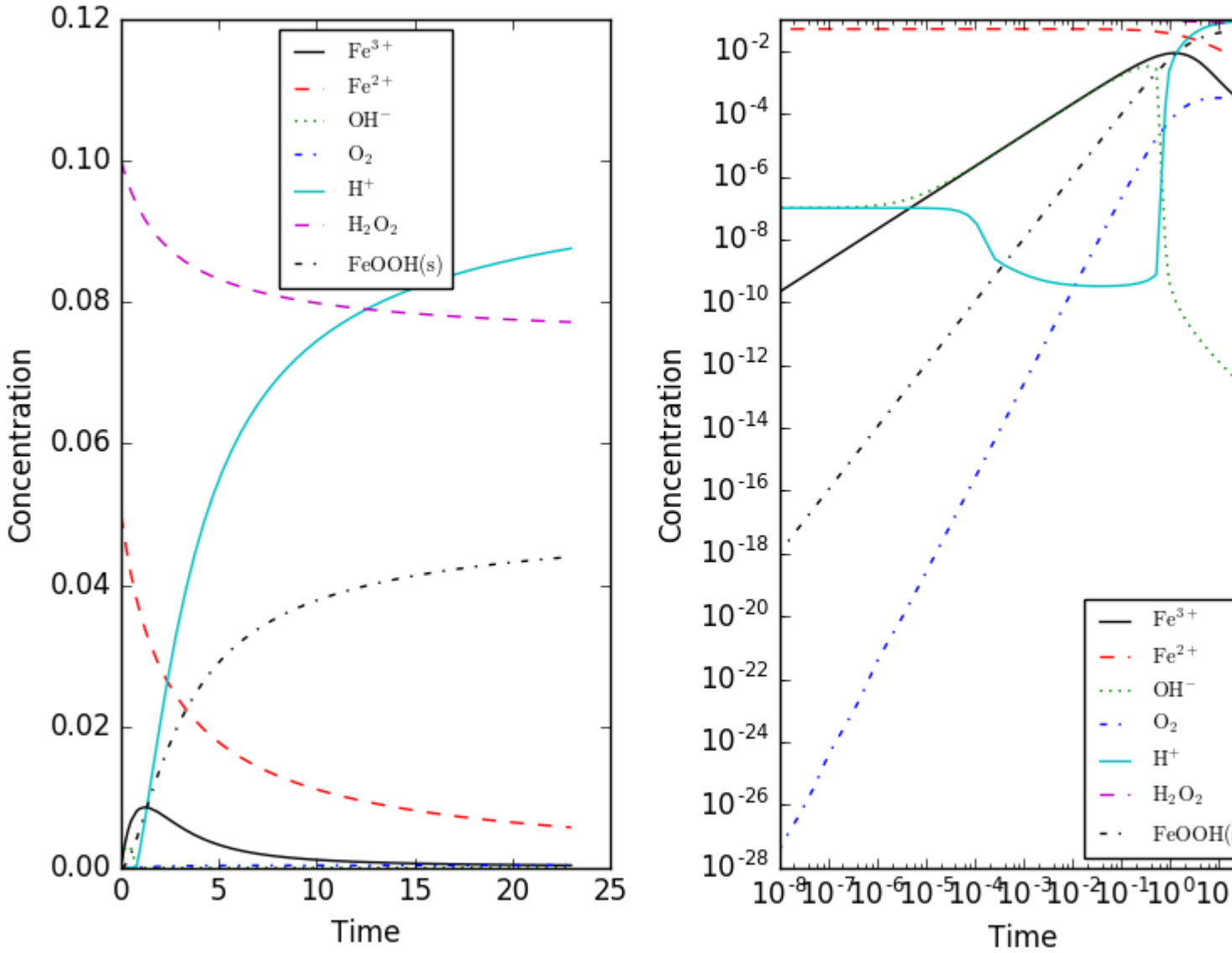
Fuerza iónica

```
 from chempy.electrolytes import ionic_strength
     ionic_strength({'Fe+3': 0.050, 'ClO4-': 0.150}) == .3
    True
```

Cinética química (sistema de ecuaciones diferenciales ordinarias)

```
from chempy import ReactionSystem  # The rate constants below are arbitrary
 rsys = ReactionSystem.from_string("""2 Fe+2 + H2O2 -> 2 Fe+3 + 2 OH-; 42
     2 Fe+3 + H2O2 -> 2 Fe+2 + O2 + 2 H+; 17
     H+ + OH- -> H2O; 1e10
     H2O -> H+ + OH-; 1e-4
     Fe+3 + 2 H2O -> FeOOH(s) + 3 H+; 1
     FeOOH(s) + 3 H+ -> Fe+3 + 2 H2O; 2.5""")  # "[H2O]" = 1.0 (actually 55.4 at RT)
 from chempy.kinetics.ode import get_odesys
 odesys, extra = get_odesys(rsys)
 from collections import defaultdict
 import numpy as np
 tout = sorted(np.concatenate((np.linspace(0, 23), np.logspace(-8, 1))))
 c0 = defaultdict(float, {'Fe+2': 0.05, 'H2O2': 0.1, 'H2O': 1.0, 'H+': 1e-7, 'OH-': 1e-7})
 result = odesys.integrate(tout, c0, atol=1e-12, rtol=1e-14)
```

```
 import matplotlib.pyplot as plt
 _ = plt.subplot(1, 2, 1)
 _ = result.plot(names=[k for k in rsys.substances if k != 'H2O'])
 _ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
 _ = plt.subplot(1, 2, 2)
 _ = result.plot(names=[k for k in rsys.substances if k != 'H2O'], xscale='log', yscale='log')
 _ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
 _ = plt.tight_layout()
 plt.show()
```



Lea ChemPy - paquete de python en línea: https://riptutorial.com/es/python/topic/10625/chempy---paquete-de-python

Capítulo 27: Clases base abstractas (abc)

Examples

Configuración de la metaclase ABCMeta

Las clases abstractas son clases que están destinadas a ser heredadas pero evitan la implementación de métodos específicos, dejando atrás solo las firmas de métodos que las subclases deben implementar.

Las clases abstractas son útiles para definir y hacer cumplir las abstracciones de clase a un alto nivel, similar al concepto de interfaces en lenguajes escritos, sin la necesidad de implementar métodos.

Un enfoque conceptual para definir una clase abstracta es eliminar los métodos de la clase y, a continuación, generar un error NotImplemented si se accede. Esto evita que las clases de niños accedan a los métodos de los padres sin anularlos primero. Al igual que:

```
class Fruit:

    def check_ripeness(self):
        raise NotImplementedError("check_ripeness method not implemented!")


class Apple(Fruit):
    pass


a = Apple()
a.check_ripeness() # raises NotImplementedError
```

La creación de una clase abstracta de esta manera evita el uso incorrecto de métodos que no se anulan, y ciertamente alienta a los métodos a ser definidos en clases secundarias, pero no impone su definición. Con el módulo `abc` podemos evitar que las clases secundarias se instalen cuando no pueden anular los métodos de clase abstracta de sus padres y ancestros:

```
from abc import ABCMeta

class AbstractClass(object):
    # the metaclass attribute must always be set as a class variable
    __metaclass__ = ABCMeta

   # the abstractmethod decorator registers this method as undefined
   @abstractmethod
   def virtual_method_subclasses_must_define(self):
       # Can be left completely blank, or a base implementation can be provided
       # Note that ordinarily a blank interpretation implicitly returns `None`,
       # but by registering, this behaviour is no longer enforced.
```

Ahora es posible simplemente subclasificar y anular:

```
class Subclass(AbstractClass):
    def virtual_method_subclasses_must_define(self):
        return
```

Por qué / Cómo usar ABCMeta y @abstractmethod

Las clases base abstractas (ABC) imponen qué clases derivadas implementan métodos particulares de la clase base.

Para entender cómo funciona esto y por qué debemos usarlo, echemos un vistazo a un ejemplo que Van Rossum disfrutaría. Digamos que tenemos una clase base "MontyPython" con dos métodos (broma y línea de remate) que deben ser implementados por todas las clases derivadas.

```
class MontyPython:
    def joke(self):
        raise NotImplementedError()

    def punchline(self):
        raise NotImplementedError()

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

Cuando creamos una instancia de un objeto y llamamos a sus dos métodos, obtendremos un error (como se esperaba) con el método `punchline()` .

```
 >>> sketch = ArgumentClinic()
 >>> sketch.punchline()
NotImplementedError
```

Sin embargo, esto todavía nos permite crear una instancia de un objeto de la clase ArgumentClinic sin obtener un error. De hecho, no recibimos un error hasta que buscamos el punchline ().

Esto se evita utilizando el módulo de clase base abstracta (ABC). Veamos cómo funciona esto con el mismo ejemplo:

```
from abc import ABCMeta, abstractmethod

class MontyPython(metaclass=ABCMeta):
    @abstractmethod
    def joke(self):
        pass

@abstractmethod
def punchline(self):
    pass

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

Esta vez, cuando intentamos crear una instancia de un objeto de la clase incompleta, ¡inmediatamente obtenemos un TypeError!

```
>>> c = ArgumentClinic()
TypeError:
"Can't instantiate abstract class ArgumentClinic with abstract methods punchline"
```

En este caso, es fácil completar la clase para evitar cualquier TypeErrors:

```
class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"

    def punchline(self):
        return "Send in the constable!"
```

¡Esta vez cuando creas un objeto, funciona!

Lea Clases base abstractas (abc) en línea: https://riptutorial.com/es/python/topic/5442/clases-base-abstractas--abc-

Capítulo 28: Clasificación, mínimo y máximo

Examples

Obteniendo el mínimo o máximo de varios valores.

```
min(7,2,1,5)
# Output: 1

max(7,2,1,5)
# Output: 7
```

Usando el argumento clave

Encontrar el mínimo / máximo de una secuencia de secuencias es posible:

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Output: (0, 10)
```

pero si desea ordenar por un elemento específico en cada secuencia, use la `key` -argumento:

```
min(list_of_tuples, key=lambda x: x[0])         # Sorting by first element
# Output: (0, 10)

min(list_of_tuples, key=lambda x: x[1])         # Sorting by second element
# Output: (2, 8)

sorted(list_of_tuples, key=lambda x: x[0])      # Sorting by first element (increasing)
# Output: [(0, 10), (1, 15), (2, 8)]

sorted(list_of_tuples, key=lambda x: x[1])      # Sorting by first element
# Output: [(2, 8), (0, 10), (1, 15)]

import operator
# The operator module contains efficient alternatives to the lambda function
max(list_of_tuples, key=operator.itemgetter(0)) # Sorting by first element
# Output: (2, 8)

max(list_of_tuples, key=operator.itemgetter(1)) # Sorting by second element
# Output: (1, 15)

sorted(list_of_tuples, key=operator.itemgetter(0), reverse=True) # Reversed (decreasing)
# Output: [(2, 8), (1, 15), (0, 10)]

sorted(list_of_tuples, key=operator.itemgetter(1), reverse=True) # Reversed(decreasing)
# Output: [(1, 15), (0, 10), (2, 8)]
```

Argumento predeterminado a max, min

No puedes pasar una secuencia vacía a `max` o `min` :

```
min([])
```

ValueError: min () arg es una secuencia vacía

Sin embargo, con Python 3, puede pasar el `default` argumento de palabra clave con un valor que se devolverá si la secuencia está vacía, en lugar de generar una excepción:

```
max([], default=42)
# Output: 42
max([], default=0)
# Output: 0
```

Caso especial: diccionarios

Obtener el mínimo o el máximo o usar `sorted` depende de las iteraciones sobre el objeto. En el caso de `dict` , la iteración es solo sobre las teclas:

```
adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Output: 'a'
max(adict)
# Output: 'c'
sorted(adict)
# Output: ['a', 'b', 'c']
```

Para mantener la estructura del diccionario, debe iterar sobre `.items()` :

```
min(adict.items())
# Output: ('a', 3)
max(adict.items())
# Output: ('c', 1)
sorted(adict.items())
# Output: [('a', 3), ('b', 5), ('c', 1)]
```

Para `sorted` , puede crear un `OrderedDict` para mantener la clasificación mientras tiene una estructura similar a un `dict` :

```
from collections import OrderedDict
OrderedDict(sorted(adict.items()))
# Output: OrderedDict([('a', 3), ('b', 5), ('c', 1)])
res = OrderedDict(sorted(adict.items()))
res['a']
# Output: 3
```

Por valor

De nuevo, esto es posible usando el argumento `key` :

```
min(adict.items(), key=lambda x: x[1])
# Output: ('c', 1)
```

```
max(adict.items(), key=operator.itemgetter(1))
# Output: ('b', 5)
sorted(adict.items(), key=operator.itemgetter(1), reverse=True)
# Output: [('b', 5), ('a', 3), ('c', 1)]
```

Obteniendo una secuencia ordenada

Usando **una** secuencia:

```
sorted((7, 2, 1, 5))                # tuple
# Output: [1, 2, 5, 7]

sorted(['c', 'A', 'b'])             # list
# Output: ['A', 'b', 'c']

sorted({11, 8, 1})                  # set
# Output: [1, 8, 11]

sorted({'11': 5, '3': 2, '10': 15})  # dict
# Output: ['10', '11', '3']          # only iterates over the keys

sorted('bdca')                      # string
# Output: ['a','b','c','d']
```

El resultado es siempre una nueva `list` ; Los datos originales se mantienen sin cambios.

Mínimo y máximo de una secuencia.

Obtener el mínimo de una secuencia (iterable) es equivalente a acceder al primer elemento de una secuencia `sorted` :

```
min([2, 7, 5])
# Output: 2
sorted([2, 7, 5])[0]
# Output: 2
```

El máximo es un poco más complicado, porque `sorted` mantiene el orden y `max` devuelve el primer valor encontrado. En caso de que no haya duplicados, el máximo es el mismo que el último elemento de la declaración ordenada:

```
max([2, 7, 5])
# Output: 7
sorted([2, 7, 5])[-1]
# Output: 7
```

Pero no si hay varios elementos que se evalúan como teniendo el valor máximo:

```
class MyClass(object):
    def __init__(self, value, name):
        self.value = value
        self.name = name

    def __lt__(self, other):
```

```
        return self.value < other.value

    def __repr__(self):
        return str(self.name)

sorted([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: [second, first, third]
max([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: first
```

Se permite cualquier elemento que contenga iterable que admita `<` o `>` operaciones.

Hacer clases personalizables ordenable

`min` , `max` y `sorted` todos necesitan que los objetos se puedan ordenar. Para ser ordenados adecuadamente, la clase necesita definir todos los 6 métodos `__lt__` , `__gt__` , `__ge__` , `__le__` , `__ne__` y `__eq__` :

```
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __le__(self, other):
        print('{!r} - Test less than or equal to {!r}'.format(self, other))
        return self.value <= other.value

    def __gt__(self, other):
        print('{!r} - Test greater than {!r}'.format(self, other))
        return self.value > other.value

    def __ge__(self, other):
        print('{!r} - Test greater than or equal to {!r}'.format(self, other))
        return self.value >= other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value
```

Aunque implementar todos estos métodos parece innecesario, omitir algunos de ellos hará que su código sea propenso a errores .

Ejemplos:

```
alist = [IntegerContainer(5), IntegerContainer(3),
         IntegerContainer(10), IntegerContainer(7)
```

```
        ]

res = max(alist)
# Out: IntegerContainer(3) - Test greater than IntegerContainer(5)
#      IntegerContainer(10) - Test greater than IntegerContainer(5)
#      IntegerContainer(7) - Test greater than IntegerContainer(10)
print(res)
# Out: IntegerContainer(10)

res = min(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
#      IntegerContainer(7) - Test less than IntegerContainer(3)
print(res)
# Out: IntegerContainer(3)

res = sorted(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
#      IntegerContainer(10) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(10)
print(res)
# Out: [IntegerContainer(3), IntegerContainer(5), IntegerContainer(7), IntegerContainer(10)]
```

`sorted` con `reverse=True` también usa `__lt__` :

```
res = sorted(alist, reverse=True)
# Out: IntegerContainer(10) - Test less than IntegerContainer(7)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(3)
print(res)
# Out: [IntegerContainer(10), IntegerContainer(7), IntegerContainer(5), IntegerContainer(3)]
```

Pero `sorted` puede usar `__gt__` cambio si el valor predeterminado no está implementado:

```
del IntegerContainer.__lt__   # The IntegerContainer no longer implements "less than"

res = min(alist)
# Out: IntegerContainer(5) - Test greater than IntegerContainer(3)
#      IntegerContainer(3) - Test greater than IntegerContainer(10)
#      IntegerContainer(3) - Test greater than IntegerContainer(7)
print(res)
# Out: IntegerContainer(3)
```

Los métodos de clasificación generarán un `TypeError` si no se implementan `__lt__` ni `__gt__` :

```
del IntegerContainer.__gt__   # The IntegerContainer no longer implements "greater then"

res = min(alist)
```

TypeError: tipos no ordenados: IntegerContainer () <IntegerContainer ()

`functools.total_ordering` decorador `functools.total_ordering` se puede utilizar para simplificar el esfuerzo de escribir estos métodos de comparación ricos. Si `total_ordering` tu clase con `total_ordering` , necesitas implementar `__eq__` , `__ne__` y solo uno de los `__lt__` , `__le__` , `__ge__` o `__gt__` , y el decorador completará el resto:

```
import functools

@functools.total_ordering
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value


IntegerContainer(5) > IntegerContainer(6)
# Output: IntegerContainer(5) - Test less than IntegerContainer(6)
# Returns: False

IntegerContainer(6) > IntegerContainer(5)
# Output: IntegerContainer(6) - Test less than IntegerContainer(5)
# Output: IntegerContainer(6) - Test equal to IntegerContainer(5)
# Returns True
```

Observe cómo el `>` (*mayor que*) ahora llama al método *menos que* , y en algunos casos incluso el método `__eq__` . Esto también significa que si la velocidad es de gran importancia, debe implementar cada método de comparación enriquecida.

Extraer N artículos más grandes o N más pequeños de un iterable

Para encontrar un número (más de uno) de los valores más grandes o más pequeños de un iterable, puede usar el `nlargest` y `nsmallest` del módulo `heapq` :

```
import heapq

# get 5 largest items from the range

heapq.nlargest(5, range(10))
# Output: [9, 8, 7, 6, 5]

heapq.nsmallest(5, range(10))
# Output: [0, 1, 2, 3, 4]
```

Esto es mucho más eficiente que clasificar todo el material y luego cortarlo desde el final o el principio. Internamente, estas funciones utilizan la estructura de datos de la cola de prioridad del montón binario , que es muy eficiente para este caso de uso.

Al igual que `min` , `max` y `sorted` , estas funciones aceptan el argumento de palabra `key` clave opcional, que debe ser una función que, dado un elemento, devuelve su clave de clasificación.

Aquí hay un programa que extrae 1000 líneas más largas de un archivo:

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

Aquí abrimos el archivo y pasamos el identificador de archivo `f` a `nlargest` . La iteración del archivo produce cada línea del archivo como una cadena separada; `nlargest` luego pasa cada elemento (o línea) que pasa a la función `len` para determinar su clave de clasificación. `len` , dada una cadena, devuelve la longitud de la línea en caracteres.

Esto solo necesita almacenamiento para una lista de 1000 líneas más grandes hasta el momento, que se puede contrastar con

```
longest_lines = sorted(f, key=len)[1000:]
```

que tendrá que mantener *todo el archivo en la memoria* .

Lea Clasificación, mínimo y máximo en línea:
https://riptutorial.com/es/python/topic/252/clasificacion--minimo-y-maximo

Capítulo 29: Comentarios y Documentación

Sintaxis

- # Este es un comentario de una sola línea.
- imprimir ("") # Este es un comentario en línea
- """
  Esto es
  un comentario multilínea
  """

Observaciones

Los desarrolladores deben seguir las pautas PEP257 - Docstring Convenciones . En algunos casos, las guías de estilo (como las de la Guía de estilo de Google) o la documentación que muestra a terceros (como Sphinx) pueden detallar convenciones adicionales para las cadenas de documentación.

Examples

Comentarios de línea única, en línea y multilínea.

Los comentarios se utilizan para explicar el código cuando el código básico en sí no está claro.

Python ignora los comentarios, por lo que no ejecutará el código allí, ni generará errores de sintaxis para las oraciones simples en inglés.

Los comentarios de una sola línea comienzan con el carácter de hash (`#`) y terminan al final de la línea.

- Comentario de una sola línea:

```
# This is a single line comment in Python
```

- Comentario en línea:

```
print("Hello World")  # This line prints "Hello World"
```

- Los comentarios que abarcan varias líneas tienen `"""` o `'''` en cualquiera de los extremos. Es lo mismo que una cadena multilínea, pero se pueden usar como comentarios:

```
"""
This type of comment spans multiple lines.
These are mostly used for documentation of functions, classes and modules.
"""
```

Accediendo programáticamente a las cadenas de documentación

Las cadenas de documentos son, a diferencia de los comentarios regulares, almacenados como un atributo de la función que documentan, lo que significa que puede acceder a ellos mediante programación.

Una función de ejemplo

```
def func():
    """This is a function that does nothing at all"""
    return
```

Se puede acceder a la `__doc__` utilizando el atributo `__doc__` :

```
print(func.__doc__)
```

Esta es una función que no hace nada en absoluto.

```
help(func)
```

Ayuda en la función de `func` en el módulo `__main__` :

```
func()
```

Esta es una función que no hace nada en absoluto.

Otra función de ejemplo

`function.__doc__` es solo la cadena de documentos real como una cadena, mientras que la función de `help` proporciona información general sobre una función, incluida la cadena de documentos. Aquí hay un ejemplo más útil:

```
def greet(name, greeting="Hello"):
    """Print a greeting to the user `name`

    Optional parameter `greeting` can change what they're greeted with."""

    print("{} {}".format(greeting, name))
```

```
help(greet)
```

Ayuda en función `greet` en el módulo `__main__` :

```
greet(name, greeting='Hello')
```

Imprime un saludo al `name` usuario
El parámetro de `greeting` opcional puede cambiar lo que se saluda con.

Ventajas de docstrings sobre comentarios regulares

El solo hecho de no poner una cadena de documentos o un comentario regular en una función hace que sea mucho menos útil.

```
def greet(name, greeting="Hello"):
    # Print a greeting to the user `name`
    # Optional parameter `greeting` can change what they're greeted with.

    print("{} {}".format(greeting, name))
```

```
print(greet.__doc__)
```

Ninguna

```
help(greet)
```

Ayuda en función saludar en módulo **principal** :

```
greet(name, greeting='Hello')
```

Escribir documentación utilizando cadenas de documentación.

Una cadena de documentación es un comentario de varias líneas que se utiliza para documentar módulos, clases, funciones y métodos. Tiene que ser la primera declaración del componente que describe.

```
def hello(name):
    """Greet someone.

    Print a greeting ("Hello") for the person with the given name.
    """

    print("Hello "+name)
```

```
class Greeter:
    """An object used to greet people.

    It contains multiple greeting functions for several languages
    and times of the  day.
    """
```

Se puede acceder al valor de la cadena de documentos dentro del programa y, por ejemplo, el comando de `help` utiliza.

Convenciones de sintaxis

PEP 257

PEP 257 define un estándar de sintaxis para comentarios de cadena de documentación. Básicamente permite dos tipos:

- Docstrings de una línea:

Según PEP 257, deben usarse con funciones cortas y simples. Todo se coloca en una línea, por ejemplo:

```
def hello():
    """Say hello to your friends."""
    print("Hello my friends!")
```

La cadena de documentos debe terminar con un punto, el verbo debe estar en la forma imperativa.

- Docstrings multilínea:

La cadena de documentos multilínea se debe utilizar para funciones, módulos o clases más largas y complejas.

```
def hello(name, language="en"):
    """Say hello to a person.

    Arguments:
    name: the name of the person
    language: the language in which the person should be greeted
    """

    print(greeting[language]+" "+name)
```

Comienzan con un breve resumen (equivalente al contenido de una cadena de documentación de una línea) que puede estar en la misma línea que las comillas o en la siguiente línea, dan detalles adicionales y enumeran parámetros y valores de retorno.

La nota PEP 257 define qué información se debe dar dentro de una cadena de documentación, no define en qué formato se debe dar. Esta fue la razón para que otras partes y herramientas de análisis de documentación especifiquen sus propios estándares para la documentación, algunos de los cuales se enumeran a continuación y en esta pregunta .

Esfinge

Sphinx es una herramienta para generar documentación basada en HTML para proyectos de Python basados en cadenas de documentación. Su lenguaje de marcado utilizado es reStructuredText . Definen sus propios estándares para la documentación, pythonhosted.org alberga una muy buena descripción de ellos . El formato Sphinx es, por ejemplo, utilizado por el IDE de pyCharm .

Una función se documentaría así utilizando el formato Sphinx / reStructuredText:

```
def hello(name, language="en"):
```

```
    """Say hello to a person.

    :param name: the name of the person
    :type name: str
    :param language: the language in which the person should be greeted
    :type language: str
    :return: a number
    :rtype: int
    """

    print(greeting[language]+" "+name)
    return 4
```

Guía de estilo de Google Python

Google ha publicado la Guía de estilo de Google Python que define las convenciones de codificación para Python, incluidos los comentarios de la documentación. En comparación con Sphinx / reST, muchas personas dicen que la documentación de acuerdo con las directrices de Google es mejor legible para los humanos.

La página pythonhosted.org mencionada anteriormente también proporciona algunos ejemplos de buena documentación de acuerdo con la Guía de estilo de Google.

Al usar el complemento de Napoleón , Sphinx también puede analizar la documentación en el formato compatible con la Guía de estilo de Google.

Una función se documentaría así utilizando el formato de la Guía de estilo de Google:

```
def hello(name, language="en"):
    """Say hello to a person.

    Args:
        name: the name of the person as string
        language: the language code string

    Returns:
        A number.
    """

    print(greeting[language]+" "+name)
    return 4
```

Lea Comentarios y Documentación en línea:
https://riptutorial.com/es/python/topic/4144/comentarios-y-documentacion

Capítulo 30: Comparaciones

Sintaxis

- ! = - No es igual a
- == - es igual a
- `>` - mayor que
- `<` - menos que
- `>=` - mayor que o igual a
- `<=` - menor o igual que
- `is` - prueba si los objetos son exactamente el mismo objeto
- no es = prueba si los objetos no son exactamente el mismo objeto

Parámetros

Parámetro	Detalles
X	Primer artículo a comparar
y	Segundo elemento a comparar

Examples

Mayor o menor que

```
x > y
x < y
```

Estos operadores comparan dos tipos de valores, son menos que y mayores que los operadores. Para los números, esto simplemente compara los valores numéricos para ver cuál es más grande:

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

Para las cuerdas, se compararán lexicográficamente, lo cual es similar al orden alfabético pero no

es exactamente el mismo.

```
"alpha" < "beta"
# True
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

En estas comparaciones, las letras minúsculas se consideran "mayores que" en mayúsculas, por lo que `"gamma" < "OMEGA"` es falso. Si todos estuvieran en mayúsculas, devolvería el resultado de orden alfabético esperado:

```
"GAMMA" < "OMEGA"
# True
```

Cada tipo define su cálculo con los operadores `<` y `>` diferente, por lo que debe investigar qué significan los operadores con un tipo dado antes de usarlo.

No igual a

```
x != y
```

Esto devuelve `True` si `x` e `y` no son iguales y, de lo contrario, devuelve `False` .

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

Igual a

```
x == y
```

Esta expresión se evalúa si `x` y `y` son del mismo valor y devuelve el resultado como un valor booleano. En general, tanto el tipo como el valor deben coincidir, por lo que el int `12` no es lo mismo que la cadena `'12'` .

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
# True
'spam' == 'spam '
# False
'12' == 12
```

```
# False
```

Tenga en cuenta que cada tipo debe definir una función que se utilizará para evaluar si dos valores son iguales. Para los tipos incorporados, estas funciones se comportan como cabría esperar y solo evalúan las cosas basándose en el mismo valor. Sin embargo, los tipos personalizados pueden definir las pruebas de igualdad como lo deseen, incluyendo devolver siempre `True` o siempre `False` .

Comparaciones de cadena

Puede comparar varios elementos con múltiples operadores de comparación con comparación de cadena. Por ejemplo

```
x > y > z
```

Es sólo una forma corta de:

```
x > y and y > z
```

Esto se evaluará como `True` solo si ambas comparaciones son `True` .

La forma general es

```
a OP b OP c OP d ...
```

Donde `OP` representa una de las múltiples operaciones de comparación que puede usar, y las letras representan expresiones válidas arbitrarias.

> Tenga en cuenta que `0 != 1 != 0` evalúa como `True` , aunque `0 != 0` sea `False` . A diferencia de la notación matemática común en la que `x != y != z` significa que `x` , `y` , `z` tienen valores diferentes. Las operaciones de encadenamiento `==` tienen el significado natural en la mayoría de los casos, ya que la igualdad es generalmente transitiva.

Estilo

No hay un límite teórico sobre la cantidad de elementos y operaciones de comparación que utiliza, siempre que tenga la sintaxis adecuada:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

Lo anterior devuelve `True` si cada comparación devuelve `True` . Sin embargo, el uso de encadenamiento enrevesado no es un buen estilo. Un buen encadenamiento será "direccional", no más complicado que

```
1 > x > -4 > y != 8
```

Efectos secundarios

Tan pronto como una comparación devuelve `False` , la expresión se evalúa inmediatamente en `False` , omitiendo todas las comparaciones restantes.

Tenga en cuenta que la expresión `exp` en `a > exp > b` se evaluará solo una vez, mientras que en el caso de

```
a > exp and exp > b
```

`exp` se calculará dos veces si `a > exp` es verdadero.

Comparación por `is` vs` == `

Un error común es confundir a los operadores de comparación de igualdad `is` y `==` .

`a == b` compara el valor de `a` y `b` .

`a is b` comparará las *identidades* de `a` y `b` .

Para ilustrar:

```
a = 'Python is fun!'
b = 'Python is fun!'
a == b # returns True
a is b # returns False

a = [1, 2, 3, 4, 5]
b = a      # b references a
a == b     # True
a is b     # True
b = a[:]   # b now references a copy of a
a == b     # True
a is b     # False [!!]
```

Básicamente, `is` puede considerar como una abreviatura para `id(a) == id(b)` .

Más allá de esto, hay peculiaridades del entorno del tiempo de ejecución que complican aún más las cosas. Las cadenas cortas y los enteros pequeños devolverán `True` cuando se compara con `is` , debido a que la máquina Python intenta usar menos memoria para objetos idénticos.

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

Pero las cadenas más largas y los enteros más grandes se almacenarán por separado.

```
a = 'not so short'
b = 'not so short'
c = 1000
d = 1000
a is b # False
c is d # False
```

Usted debe usar `is` para probar para `None` :

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

Un uso de `is` es probar un "centinela" (es decir, un objeto único).

```
sentinel = object()
def myfunc(var=sentinel):
    if var is sentinel:
        # value wasn't provided
        pass
    else:
        # value was provided
        pass
```

Comparando objetos

Para comparar la igualdad de clases personalizadas, puede anular `==` y `!=` `__ne__` métodos `__eq__` y `__ne__` . También puede anular `__lt__` (`<`), `__le__` (`<=`), `__gt__` (`>`) y `__ge__` (`>`). Tenga en cuenta que solo necesita anular dos métodos de comparación, y Python puede manejar el resto (`==` es lo mismo que `not <` y `not >` , etc.)

```
class Foo(object):
    def __init__(self, item):
        self.my_item = item
    def __eq__(self, other):
        return self.my_item == other.my_item

a = Foo(5)
b = Foo(5)
a == b     # True
a != b     # False
a is b     # False
```

Tenga en cuenta que esta comparación simple supone que `other` (el objeto que se está comparando) es el mismo tipo de objeto. Comparando con otro tipo lanzará un error:

```
class Bar(object):
    def __init__(self, item):
        self.other_item = item
    def __eq__(self, other):
        return self.other_item == other.other_item
```

```
    def __ne__(self, other):
        return self.other_item != other.other_item

c = Bar(5)
a == c    # throws AttributeError: 'Foo' object has no attribute 'other_item'
```

Verificar `isinstance()` o similar ayudará a prevenir esto (si se desea).

Common Gotcha: Python no impone la escritura

En muchos otros idiomas, si ejecuta lo siguiente (ejemplo de Java)

```
if("asgdsrf" == 0) {
    //do stuff
}
```

... obtendrá un error. No puedes ir comparando cadenas con enteros como ese. En Python, esta es una declaración perfectamente legal, solo se resolverá en `False` .

Un gotcha común es el siguiente

```
myVariable = "1"
if 1 == myVariable:
    #do stuff
```

Esta comparación evaluará a `False` sin un error, cada vez, ocultando un error o rompiendo un condicional.

Lea Comparaciones en línea: https://riptutorial.com/es/python/topic/248/comparaciones

Capítulo 31: Complementos y clases de extensión

Examples

Mixins

En el lenguaje de programación orientado a objetos, una mezcla es una clase que contiene métodos para el uso de otras clases sin tener que ser la clase principal de esas otras clases. Cómo esas otras clases obtienen acceso a los métodos de la mezcla depende del idioma.

Proporciona un mecanismo para la herencia múltiple al permitir que varias clases usen la funcionalidad común, pero sin la semántica compleja de la herencia múltiple. Los mixins son útiles cuando un programador quiere compartir funcionalidad entre diferentes clases. En lugar de repetir el mismo código una y otra vez, la funcionalidad común puede simplemente agruparse en una mezcla y luego heredarse en cada clase que lo requiera.

Cuando usamos más de un mixins, el orden de los mixins es importante. Aquí hay un ejemplo simple:

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(Mixin1, Mixin2):
    pass
```

En este ejemplo llamamos `MyClass` y método de `test` ,

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

El resultado debe ser Mixin1 porque el orden es de izquierda a derecha. Esto podría mostrar resultados inesperados cuando las súper clases lo agreguen. Así que el orden inverso es más bueno así:

```
class MyClass(Mixin2, Mixin1):
    pass
```

El resultado será:

```
>>> obj = MyClass()
```

```
>>> obj.test()
Mixin2
```

Los mixins se pueden utilizar para definir complementos personalizados.

Python 3.x 3.0

```
class Base(object):
    def test(self):
        print("Base.")

class PluginA(object):
    def test(self):
        super().test()
        print("Plugin A.")

class PluginB(object):
    def test(self):
        super().test()
        print("Plugin B.")

plugins = PluginA, PluginB

class PluginSystemA(PluginA, Base):
    pass

class PluginSystemB(PluginB, Base):
    pass

PluginSystemA().test()
# Base.
# Plugin A.

 PluginSystemB().test()
# Base.
# Plugin B.
```

Plugins con clases personalizadas

En Python 3.6, PEP 487 agregó el método especial `__init_subclass__` , que simplifica y amplía la personalización de la clase sin usar metaclases . En consecuencia, esta característica permite crear complementos simples . Aquí demostramos esta característica modificando un ejemplo anterior :

Python 3.x 3.6

```
class Base:
    plugins = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.plugins.append(cls)

    def test(self):
        print("Base.")

class PluginA(Base):
```

```
    def test(self):
        super().test()
        print("Plugin A.")


class PluginB(Base):
    def test(self):
        super().test()
        print("Plugin B.")
```

Resultados:

```
PluginA().test()
# Base.
# Plugin A.

PluginB().test()
# Base.
# Plugin B.

Base.plugins
# [__main__.PluginA, __main__.PluginB]
```

Lea Complementos y clases de extensión en línea:
https://riptutorial.com/es/python/topic/4724/complementos-y-clases-de-extension

Capítulo 32: Comprobando la existencia de ruta y permisos

Parámetros

Parámetro	Detalles
os.F_OK	Valor para pasar como el parámetro de modo de acceso () para probar la existencia de ruta.
os.R_OK	Valor para incluir en el parámetro de modo de acceso () para probar la legibilidad de la ruta.
os.W_OK	Valor para incluir en el parámetro de modo de acceso () para probar la capacidad de escritura de la ruta.
os.X_OK	Valor que se incluirá en el parámetro de modo de acceso () para determinar si se puede ejecutar la ruta.

Examples

Realizar comprobaciones utilizando os.access

`os.access` es una solución mucho mejor para verificar si existe un directorio y es accesible para leer y escribir.

```
import os
path = "/home/myFiles/directory1"

## Check if path exists
os.access(path, os.F_OK)

## Check if path is Readable
os.access(path, os.R_OK)

## Check if path is Wriable
os.access(path, os.W_OK)

## Check if path is Execuatble
os.access(path, os.E_OK)
```

También es posible realizar todos los controles juntos.

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.E_OK)
```

Todo lo anterior devuelve `True` si el acceso está permitido y `False` si no está permitido. Estos están

disponibles en Unix y Windows.

Lea Comprobando la existencia de ruta y permisos en línea:
https://riptutorial.com/es/python/topic/1262/comprobando-la-existencia-de-ruta-y-permisos

Capítulo 33: Computación paralela

Observaciones

Debido al GIL (bloqueo de intérprete global), solo una instancia del intérprete de python se ejecuta en un solo proceso. Por lo tanto, en general, el uso de subprocesos múltiples solo mejora los cálculos de E / S enlazados, no los de CPU. Se recomienda el módulo de `multiprocessing` si desea paralelizar las tareas relacionadas con la CPU.

GIL se aplica a CPython, la implementación más popular de Python, así como a PyPy. Otras implementaciones como Jython y IronPython no tienen GIL .

Examples

Uso del módulo multiprocesamiento para paralelizar tareas.

```
import multiprocessing

def fib(n):
    """computing the Fibonacci in an inefficient way
    was chosen to slow down the CPU."""
    if n <= 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
p = multiprocessing.Pool()
print(p.map(fib,[38,37,36,35,34,33]))

# Out: [39088169, 24157817, 14930352, 9227465, 5702887, 3524578]
```

A medida que la ejecución de cada llamada a `fib` ocurre en paralelo, el tiempo de ejecución del ejemplo completo es **1.8 × más rápido** que si se hiciera de forma secuencial en un procesador dual.

Python 2.2+

Usando scripts de Padres e Hijos para ejecutar código en paralelo

niño.py

```
import time

def main():
    print "starting work"
    time.sleep(1)
    print "work work work work work"
    time.sleep(1)
    print "done working"

if __name__ == '__main__':
```

```
    main()
```

parent.py

```
import os

def main():
    for i in range(5):
        os.system("python child.py &")

if __name__ == '__main__':
    main()
```

Esto es útil para tareas de solicitud / respuesta HTTP independientes o para selección / inserción de base de datos. Los argumentos de la línea de comando también se pueden dar al script **child.py** . La sincronización entre scripts se puede lograr si todos los scripts revisan regularmente un servidor separado (como una instancia de Redis).

Usando una extensión C para paralelizar tareas

La idea aquí es mover los trabajos intensivos en computación a C (usando macros especiales), independientemente de Python, y hacer que el código C libere el GIL mientras está funcionando.

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

Usando el módulo PyPar para paralelizar

PyPar es una biblioteca que utiliza la interfaz de paso de mensajes (MPI) para proporcionar paralelismo en Python. Un ejemplo simple en PyPar (como se ve en https://github.com/daleroberts/pypar) se ve así:

```
import pypar as pp

ncpus = pp.size()
rank = pp.rank()
node = pp.get_processor_name()

print 'I am rank %d of %d on node %s' % (rank, ncpus, node)

if rank == 0:
  msh = 'P0'
  pp.send(msg, destination=1)
  msg = pp.receive(source=rank-1)
  print 'Processor 0 received message "%s" from rank %d' % (msg, rank-1)
```

```
else:
  source = rank-1
  destination = (rank+1) % ncpus
  msg = pp.receive(source)
  msg = msg + 'P' + str(rank)
  pypar.send(msg, destination)
pp.finalize()
```

Lea Computación paralela en línea: https://riptutorial.com/es/python/topic/542/computacion-paralela

Capítulo 34: Comunicación Serial Python (pyserial)

Sintaxis

- ser.read (tamaño = 1)
- ser.readline ()
- ser.write ()

Parámetros

parámetro	detalles
Puerto	Nombre del dispositivo, por ejemplo, / dev / ttyUSB0 en GNU / Linux o COM3 en Windows.
velocidad de transmisión	tipo de baudios: int por defecto: 9600 valores estándar: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200

Observaciones

Para más detalles echa un vistazo a la documentación de pyserial.

Examples

Inicializar dispositivo serie

```
import serial
#Serial takes these two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

Leer del puerto serial

Inicializar dispositivo serie

```
import serial
#Serial takes two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

para leer un solo byte desde el dispositivo serie

```
data = ser.read()
```

para leer el número dado de bytes del dispositivo serie

```
data = ser.read(size=5)
```

para leer una línea desde el dispositivo serie.

```
data = ser.readline()
```

para leer los datos del dispositivo serie mientras se escribe algo sobre él.

```
#for python2.7
data = ser.read(ser.inWaiting())

#for python3
ser.read(ser.inWaiting)
```

Compruebe qué puertos serie están disponibles en su máquina

Para obtener una lista de los puertos serie disponibles use

```
python -m serial.tools.list_ports
```

en un símbolo del sistema o

```
from serial.tools import list_ports
list_ports.comports()  # Outputs list of available serial ports
```

de la cáscara de Python.

Lea Comunicación Serial Python (pyserial) en línea:
https://riptutorial.com/es/python/topic/5744/comunicacion-serial-python--pyserial-

Capítulo 35: Concurrencia de Python

Observaciones

Los desarrolladores de Python se aseguraron de que la API entre `threading` y `multiprocessing` sea similar, de modo que el cambio entre las dos variantes sea más fácil para los programadores.

Examples

El módulo de enhebrado.

```
from __future__ import print_function
import threading
def counter(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
t2 = threading.Thread(target=countdown,args=(20,))
t2.start()
```

En ciertas implementaciones de Python como CPython, cierto paralelismo no se consigue utilizando hilos porque de la utilización de lo que se conoce como el GIL, o **G** lobal **I** nterpreter **L** ock.

Aquí hay una excelente descripción de la concurrencia de Python:

Concordancia Python por David Beazley (YouTube)

El módulo multiprocesamiento.

```
from __future__ import print_function
import multiprocessing


def countdown(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=countdown, args=(10,))
    p1.start()

    p2 = multiprocessing.Process(target=countdown, args=(20,))
    p2.start()
```

```
    p1.join()
    p2.join()
```

Aquí, cada función se ejecuta en un nuevo proceso. Dado que una nueva instancia de Python VM ejecuta el código, no hay `GIL` y se ejecuta el paralelismo en varios núcleos.

El método `Process.start` inicia este nuevo proceso y ejecuta la función pasada en el argumento de `target` con los argumentos `args` . El método `Process.join` espera el final de la ejecución de los procesos `p1` y `p2` .

Los nuevos procesos se inician de manera diferente según la versión de python y la plataforma en la que se ejecuta el código, *por ejemplo* :

- Windows usa `spawn` para crear el nuevo proceso.
- Con los sistemas y versiones de Unix anteriores a 3.3, los procesos se crean utilizando un `fork` .
  Tenga en cuenta que este método no respeta el uso POSIX de la bifurcación y, por lo tanto, conduce a comportamientos inesperados, especialmente al interactuar con otras bibliotecas de multiprocesamiento.
- Con el sistema Unix y la versión 3.4+, puede elegir iniciar los nuevos procesos con `fork` , `forkserver` o `spawn` utilizando `multiprocessing.set_start_method` al comienzo de su programa. `forkserver` métodos `forkserver` y `spawn` son más lentos que los forking, pero evitan algunos comportamientos inesperados.

Uso de la horquilla POSIX :

> Después de una bifurcación en un programa multiproceso, el niño solo puede llamar de forma segura a funciones async-signal-safe hasta el momento en que lo llame execve.
> (ver)

Usando fork, se iniciará un nuevo proceso con el mismo estado exacto para todo el mutex actual, pero solo se `MainThread` . Esto no es seguro ya que podría conducir a condiciones de carrera, *por ejemplo* :

- Si utiliza un `Lock` en `MainThread` y lo pasa a otro hilo que se supone que debe bloquearlo en algún momento. Si la `fork` ocurre simultáneamente, el nuevo proceso comenzará con un bloqueo bloqueado que nunca se liberará ya que el segundo hilo no existe en este nuevo proceso.

En realidad, este tipo de comportamiento no debería ocurrir en Python puro, ya que el `multiprocessing` maneja correctamente, pero si está interactuando con otra biblioteca, puede ocurrir este tipo de comportamiento, lo que puede provocar un fallo en su sistema (por ejemplo, con numpy / acelerado en macOS)

Transferencia de datos entre procesos de multiprocesamiento.

Debido a que los datos son confidenciales cuando se manejan entre dos subprocesos (piense que la lectura concurrente y la escritura concurrente pueden entrar en conflicto entre sí, causando

condiciones de carrera), se creó un conjunto de objetos únicos para facilitar la transferencia de datos entre los subprocesos. Cualquier operación verdaderamente atómica puede usarse entre subprocesos, pero siempre es seguro mantener la cola.

```
import multiprocessing
import queue
my_Queue=multiprocessing.Queue()
#Creates a queue with an undefined maximum size
#this can be dangerous as the queue becomes increasingly large
#it will take a long time to copy data to/from each read/write thread
```

La mayoría de las personas sugerirán que al usar la cola, siempre coloque los datos de la cola en un intento: excepto: bloque en lugar de usar vacío. Sin embargo, para las aplicaciones en las que no importa si omite un ciclo de exploración (los datos se pueden colocar en la cola mientras se `queue.Empty==True` estados de la `queue.Empty==True` a la `queue.Empty==False`) por lo general es mejor colocar la lectura y acceso de escritura en lo que yo llamo un bloque Iftry, porque una declaración 'if' es técnicamente más eficaz que atrapar la excepción.

```
import multiprocessing
import queue
'''Import necessary Python standard libraries, multiprocessing for classes and queue for the
queue exceptions it provides'''
def Queue_Iftry_Get(get_queue, default=None, use_default=False, func=None, use_func=False):
    '''This global method for the Iftry block is provided for it's reuse and
standard functionality, the if also saves on performance as opposed to catching
 the exception, which is expencive.
        It also allows the user to specify a function for the outgoing data to use,
 and a default value to return if the function cannot return the value from the queue'''
        if get_queue.empty():
            if use_default:
                return default
        else:
            try:
                value = get_queue.get_nowait()
            except queue.Empty:
                if use_default:
                    return default
            else:
                if use_func:
                    return func(value)
                else:
                    return value
    def Queue_Iftry_Put(put_queue, value):
        '''This global method for the Iftry block is provided because of its reuse
and
standard functionality, the If also saves on performance as opposed to catching
 the exception, which is expensive.
        Return True if placing value in the queue was successful. Otherwise, false'''
        if put_queue.full():
            return False
        else:
            try:
                put_queue.put_nowait(value)
            except queue.Full:
                return False
            else:
                return True
```

Lea Concurrencia de Python en línea: https://riptutorial.com/es/python/topic/3357/concurrencia-de-python

Capítulo 36: Condicionales

Introducción

Las expresiones condicionales, que incluyen palabras clave como if, elif y else, proporcionan a los programas de Python la capacidad de realizar diferentes acciones dependiendo de una condición booleana: Verdadero o Falso. Esta sección cubre el uso de condicionales de Python, lógica booleana y sentencias ternarias.

Sintaxis

- <expression> if <conditional> else <expression> # Operador ternario

Examples

si, elif, y si no

En Python puedes definir una serie de condicionales usando `if` para el primero, `elif` para el resto, hasta el final (opcional) `else` para cualquier cosa que no sea capturada por los otros condicionales.

```
number = 5

if number > 2:
    print("Number is bigger than 2.")
elif number < 2:  # Optional clause (you can have multiple elifs)
    print("Number is smaller than 2.")
else:  # Optional clause (you can only have one else)
    print("Number is 2.")
```

El `Number is bigger than 2` salidas `Number is bigger than 2`

El uso de `else if` lugar de `elif` activará un error de sintaxis y no está permitido.

Expresión condicional (o "El operador ternario")

El operador ternario se utiliza para expresiones condicionales en línea. Se utiliza mejor en operaciones simples y concisas que se leen fácilmente.

- El orden de los argumentos es diferente de muchos otros lenguajes (como C, Ruby, Java, etc.), lo que puede provocar errores cuando las personas que no están familiarizadas con el comportamiento "sorprendente" de Python lo usan (pueden invertir el orden).
- Algunos lo consideran "poco manejable", ya que va en contra del flujo normal de pensamiento (pensando primero en la condición y luego en los efectos).

```
n = 5
```

```
"Greater than 2" if n > 2 else "Smaller than or equal to 2"
# Out: 'Greater than 2'
```

El resultado de esta expresión será tal como se lee en inglés; si la expresión condicional es Verdadero, se evaluará la expresión en el lado izquierdo, de lo contrario, el lado derecho.

Las operaciones de tenencia también se pueden anidar, como aquí:

```
n = 5
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

También proporcionan un método para incluir condicionales en las funciones lambda .

Si declaración

```
if condition:
    body
```

Las declaraciones `if` comprueban la condición. Si se evalúa como `True` , ejecuta el cuerpo de la sentencia `if` . Si se evalúa como `False` , se salta el cuerpo.

```
if True:
    print "It is true!"
>> It is true!

if False:
    print "This won't get printed.."
```

La condición puede ser cualquier expresión válida:

```
if 2 + 2 == 4:
    print "I know math!"
>> I know math!
```

Otra declaración

```
if condition:
    body
else:
    body
```

La sentencia else ejecutará su cuerpo solo si las sentencias condicionales anteriores se evalúan como Falso.

```
if True:
    print "It is true!"
else:
    print "This won't get printed.."
```

```
# Output: It is true!

if False:
    print "This won't get printed.."
else:
    print "It is false!"

# Output: It is false!
```

Expresiones lógicas booleanas

Las expresiones lógicas booleanas, además de evaluar a `True` o `False` , devuelven el *valor* que se interpretó como `True` o `False` . Es una forma en Pythonic de representar la lógica que, de lo contrario, podría requerir una prueba if-else.

Y operador

El operador `and` evalúa todas las expresiones y devuelve la última expresión si todas las expresiones se evalúan como `True` . De lo contrario, devuelve el primer valor que se evalúa como `False` :

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

O operador

El operador `or` evalúa las expresiones de izquierda a derecha y devuelve el primer valor que se evalúa como `True` o el último valor (si ninguno es `True`).

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

Evaluación perezosa

Cuando utilice este enfoque, recuerde que la evaluación es perezosa. Las expresiones que no requieren evaluación para determinar el resultado no se evalúan. Por ejemplo:

```
>>> def print_me():
        print('I am here!')
>>> 0 and print_me()
0
```

En el ejemplo anterior, `print_me` nunca se ejecuta porque Python puede determinar que la expresión completa es `False` cuando encuentra el `0` (`False`). Tenga esto en cuenta si `print_me` necesita ejecutarse para servir la lógica de su programa.

Pruebas para condiciones múltiples

Un error común al verificar múltiples condiciones es aplicar la lógica de manera incorrecta.

Este ejemplo está tratando de verificar si dos variables son cada una mayor que 2. La declaración se evalúa como - `if (a) and (b > 2)` . Esto produce un resultado inesperado porque `bool(a)` evalúa como `True` cuando `a` no es cero.

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')

yes
```

Cada variable debe compararse por separado.

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')

no
```

Otro error similar se comete al verificar si una variable es uno de varios valores. La declaración en este ejemplo se evalúa como - `if (a == 3) or (4) or (6)` . Esto produce un resultado inesperado porque `bool(4)` y `bool(6)` evalúan como `True`

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
```

```
...     print('no')

yes
```

Nuevamente cada comparación debe hacerse por separado

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')

no
```

Usar el operador in es la forma canónica de escribir esto.

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')

no
```

Valores de verdad

Los siguientes valores se consideran falsey, ya que se evalúan como `False` cuando se aplican a un operador booleano.

- Ninguna
- Falso
- `0` , o cualquier valor numérico equivalente a cero, por ejemplo `0L` , `0.0` , `0j`
- Secuencias vacías: `''` , `""` , `()` , `[]`
- Asignaciones vacías: `{}`
- Tipos definidos por el usuario donde los métodos `__bool__` o `__len__` devuelven 0 o `False`

Todos los demás valores en Python se evalúan como `True` .

Nota: Un error común es simplemente verificar la Falsedad de una operación que devuelve diferentes valores de Falsey donde la diferencia es importante. Por ejemplo, usar `if foo()` lugar de más explícito `if foo() is None`

Usando la función cmp para obtener el resultado de comparación de dos objetos

Python 2 incluye una función `cmp` que le permite determinar si un objeto es menor, igual o mayor que otro objeto. Esta función se puede usar para elegir una opción de una lista basada en una de esas tres opciones.

Supongamos que necesita imprimir `'greater than'` si `x > y` , `'less than'` si `x < y` e `'equal'` si `x == y`

.

```
['equal', 'greater than', 'less than', ][cmp(x,y)]

# x,y = 1,1 output: 'equal'
# x,y = 1,2 output: 'less than'
# x,y = 2,1 output: 'greater than'
```

`cmp(x,y)` devuelve los siguientes valores

Comparación	Resultado
x <y	-1
x == y	0
x> y	1

Esta función se elimina en Python 3. Puede usar la `cmp_to_key(func)` ayuda `cmp_to_key(func)` ubicada en `functools` en Python 3 para convertir las funciones de comparación antiguas en funciones clave.

Evaluación de expresiones condicionales usando listas de comprensión

Python te permite hackear las comprensiones de la lista para evaluar expresiones condicionales.

Por ejemplo,

```
[value_false, value_true][<conditional-test>]
```

Ejemplo:

```
>> n = 16
>> print [10, 20][n <= 15]
10
```

Aquí `n<=15` devuelve `False` (que equivale a 0 en Python). Entonces, lo que Python está evaluando es:

```
[10, 20][n <= 15]
==> [10, 20][False]
==> [10, 20][0]     #False==0, True==1 (Check Boolean Equivalencies in Python)
==> 10
```

Python 2.x 2.7

El método `__cmp__` incorporado devolvió 3 valores posibles: 0, 1, -1, donde cmp (x, y) devolvió 0: si los dos objetos eran iguales 1: x> y -1: x <y

Esto podría usarse con la lista de comprensión para devolver el primer elemento (es decir, índice

0), segundo (es decir, índice 1) y último (es decir, índice -1) de la lista. Dándonos un condicional de este tipo:

```
[value_equals, value_greater, value_less][<conditional-test>]
```

Finalmente, en todos los ejemplos anteriores, Python evalúa ambas ramas antes de elegir una. Para evaluar solo la rama elegida:

```
[lambda: value_false, lambda: value_true][<test>]()
```

donde agregar el `()` al final asegura que las funciones lambda solo se llamen / evalúen al final. Así, solo evaluamos la rama elegida.

Ejemplo:

```
count = [lambda:0, lambda:N+1][count==N]()
```

Probar si un objeto es Ninguno y asignarlo

A menudo querrá asignar algo a un objeto si es `None` , lo que indica que no se ha asignado. Usaremos una `aDate` .

La forma más sencilla de hacer esto es usar la prueba `is None` .

```
if aDate is None:
    aDate=datetime.date.today()
```

(Tenga en cuenta que es más Pythonic decir `is None` lugar de `== None`).

Pero esto puede optimizarse ligeramente explotando la noción de que `not None` se evaluará como `True` en una expresión booleana. El siguiente código es equivalente:

```
if not aDate:
    aDate=datetime.date.today()
```

Pero hay una forma más pitónica. El siguiente código también es equivalente:

```
aDate=aDate or datetime.date.today()
```

Esto hace una evaluación de corto circuito . Si `aDate` se inicializa y `not None` es `not None` , entonces se asigna a sí mismo sin efecto neto. Si `is None` , entonces `datetime.date.today()` se asigna a una `aDate` .

Lea Condicionales en línea: https://riptutorial.com/es/python/topic/1111/condicionales

Capítulo 37: Conectando Python a SQL Server

Examples

Conectar al servidor, crear tabla, consultar datos

Instala el paquete:

```
$ pip install pymssql
```

```
import pymssql

SERVER = "servername"
USER = "username"
PASSWORD = "password"
DATABASE = "dbname"

connection = pymssql.connect(server=SERVER, user=USER,
                password=PASSWORD, database=DATABASE)

cursor = connection.cursor() # to access field as dictionary use cursor(as_dict=True)
cursor.execute("SELECT TOP 1 * FROM TableName")
row = cursor.fetchone()

######## CREATE TABLE ########
cursor.execute("""
CREATE TABLE posts (
    post_id INT PRIMARY KEY NOT NULL,
    message TEXT,
    publish_date DATETIME
)
""")

######## INSERT DATA IN TABLE ########
cursor.execute("""
    INSERT INTO posts VALUES(1, "Hey There", "11.23.2016")
""")
# commit your work to database
connection.commit()

######## ITERATE THROUGH RESULTS  ########
cursor.execute("SELECT TOP 10 * FROM posts ORDER BY publish_date DESC")
for row in cursor:
    print("Message: " + row[1] + " | " + "Date: " + row[2])
    # if you pass as_dict=True to cursor
    # print(row["message"])

connection.close()
```

Puede hacer cualquier cosa si su trabajo está relacionado con expresiones SQL, simplemente pase estas expresiones al método de ejecución (operaciones CRUD).

Para la declaración, el procedimiento almacenado de llamada, el manejo de errores o más control de ejemplo: pymssql.org

Lea Conectando Python a SQL Server en línea:
https://riptutorial.com/es/python/topic/7985/conectando-python-a-sql-server

Capítulo 38: Conexión segura de shell en Python

Parámetros

Parámetro	Uso
nombre de host	Este parámetro le indica al host al que se debe establecer la conexión.
nombre de usuario	nombre de usuario requerido para acceder al host
Puerto	Puerto host
contraseña	contraseña para la cuenta

Examples

conexión ssh

```
from paramiko import client
ssh = client.SSHClient() # create a new SSHClient object
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) #auto-accept unknown host keys
ssh.connect(hostname, username=username, port=port, password=password) #connect with a host
stdin, stdout, stderr = ssh.exec_command(command) # submit a command to ssh
print stdout.channel.recv_exit_status() #tells the status  1 - job failed
```

Lea Conexión segura de shell en Python en línea:
https://riptutorial.com/es/python/topic/5709/conexion-segura-de-shell-en-python

Capítulo 39: configparser

Introducción

Este módulo proporciona la clase ConfigParser que implementa un lenguaje de configuración básico en archivos INI. Puede usar esto para escribir programas Python que los usuarios finales pueden personalizar fácilmente.

Sintaxis

- Cada nueva línea contiene un nuevo par de valores clave separados por el signo =
- Las teclas se pueden separar en secciones
- En el archivo INI, cada título de la sección se escribe entre paréntesis: []

Observaciones

Todos los valores de retorno de `ConfigParser.ConfigParser().get` son cadenas. Se puede convertir a tipos más comunes gracias a `eval`

Examples

Uso básico

En config.ini:

```
[DEFAULT]
debug = True
name = Test
password = password

[FILES]
path = /path/to/file
```

En Python:

```
from ConfigParser import ConfigParser
config = ConfigParser()

#Load configuration file
config.read("config.ini")

# Access the key "debug" in "DEFAULT" section
config.get("DEFAULT", "debug")
# Return 'True'

# Access the key "path" in "FILES" destion
config.get("FILES", "path")
# Return '/path/to/file'
```

Creando programáticamente el archivo de configuración.

El archivo de configuración contiene secciones, cada sección contiene claves y valores. El módulo configparser se puede usar para leer y escribir archivos de configuración. Creando el archivo de configuración: -

```
import configparser
config = configparser.ConfigParser()
config['settings']={'resolution':'320x240',
                    'color':'blue'}
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

El archivo de salida contiene la siguiente estructura

```
[settings]
resolution = 320x240
color = blue
```

Si desea cambiar un campo en particular, obtenga el campo y asigne el valor

```
settings=config['settings']
settings['color']='red'
```

Lea configparser en línea: https://riptutorial.com/es/python/topic/9186/configparser

Capítulo 40: Conjunto

Sintaxis

- empty_set = set () # inicializa un conjunto vacío
- literal_set = {'foo', 'bar', 'baz'} # construye un conjunto con 3 cadenas dentro de él
- set_from_list = set (['foo', 'bar', 'baz']) # llama a la función set para un nuevo conjunto
- set_from_iter = set (x para x en el rango (30)) # usa iterables arbitrarios para crear un conjunto
- set_from_iter = {x para x en [random.randint (0,10) para i en rango (10)]} # notación alternativa

Observaciones

Los conjuntos *no* están *ordenados* y tienen *un tiempo de búsqueda muy rápido* (amortizado O (1) si desea obtener asistencia técnica). Es genial usarlo cuando tienes una colección de cosas, el orden no importa, y buscarás muchos artículos por nombre. Si tiene más sentido buscar elementos por un número de índice, considere usar una lista en su lugar. Si el orden importa, considera una lista también.

Los conjuntos son *mutables* y, por lo tanto, no se pueden hashear, por lo que no puede usarlos como claves de diccionario o colocarlos en otros conjuntos o en cualquier otro lugar que requiera tipos de hashable. En tales casos, puede utilizar un `frozenset` inmutable.

Los elementos de un conjunto deben ser *hashable* . Esto significa que tienen un método correcto de `__hash__` , que es consistente con `__eq__` . En general, los tipos mutables, como la `list` o el `set` , no son hashables y no se pueden colocar en un conjunto. Si se encuentra con este problema, considere usar claves `dict` y inmutables.

Examples

Consigue los elementos únicos de una lista.

Digamos que tienes una lista de restaurantes, tal vez la lees de un archivo. Te preocupas por los restaurantes *únicos* en la lista. La mejor manera de obtener los elementos únicos de una lista es convertirlos en un conjunto:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
unique_restaurants = set(restaurants)
print(unique_restaurants)
# prints {'Chicken Chicken', "McDonald's", 'Burger King'}
```

Tenga en cuenta que el conjunto no está en el mismo orden que la lista original; eso es porque los conjuntos *no* están *ordenados* , al igual que `dict` s.

Esto se puede volver a transformar fácilmente en una `List` con la función de `list` incorporada de Python, dando otra lista que es la misma que la original pero sin duplicados:

```
list(unique_restaurants)
# ['Chicken Chicken', "McDonald's", 'Burger King']
```

También es común ver esto como una sola línea:

```
# Removes all duplicates and returns another list
list(set(restaurants))
```

Ahora, cualquier operación que se pueda realizar en la lista original se puede hacer de nuevo.

Operaciones en sets

con otros sets

```
# Intersection
{1, 2, 3, 4, 5}.intersection({3, 4, 5, 6})  # {3, 4, 5}
{1, 2, 3, 4, 5} & {3, 4, 5, 6}              # {3, 4, 5}

# Union
{1, 2, 3, 4, 5}.union({3, 4, 5, 6})  # {1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5} | {3, 4, 5, 6}       # {1, 2, 3, 4, 5, 6}

# Difference
{1, 2, 3, 4}.difference({2, 3, 5})  # {1, 4}
{1, 2, 3, 4} - {2, 3, 5}            # {1, 4}

# Symmetric difference with
{1, 2, 3, 4}.symmetric_difference({2, 3, 5})  # {1, 4, 5}
{1, 2, 3, 4} ^ {2, 3, 5}                      # {1, 4, 5}

# Superset check
{1, 2}.issuperset({1, 2, 3})  # False
{1, 2} >= {1, 2, 3}           # False

# Subset check
{1, 2}.issubset({1, 2, 3})  # True
{1, 2} <= {1, 2, 3}         # True

# Disjoint check
{1, 2}.isdisjoint({3, 4})  # True
{1, 2}.isdisjoint({1, 4})  # False
```

con elementos individuales

```
# Existence check
2 in {1,2,3}      # True
4 in {1,2,3}      # False
4 not in {1,2,3}  # True

# Add and Remove
s = {1,2,3}
s.add(4)        # s == {1,2,3,4}
```

```
s.discard(3)    # s == {1,2,4}
s.discard(5)    # s == {1,2,4}

s.remove(2)     # s == {1,4}
s.remove(2)     # KeyError!
```

Las operaciones de configuración devuelven conjuntos nuevos, pero tienen las versiones locales correspondientes:

método	operación en el lugar	método en el lugar
Unión	s | = t	actualizar
intersección	s & = t	intersection_update
diferencia	s - = t	diferencia_update
diferencia de simetría	s ^ = t	symmetric_difference_update

Por ejemplo:

```
s = {1, 2}
s.update({3, 4})   # s == {1, 2, 3, 4}
```

Conjuntos versus multisets

Los conjuntos son colecciones desordenadas de elementos distintos. Pero a veces queremos trabajar con colecciones desordenadas de elementos que no son necesariamente distintos y hacer un seguimiento de las multiplicidades de los elementos.

Considera este ejemplo:

```
>>> setA = {'a','b','b','c'}
>>> setA
set(['a', 'c', 'b'])
```

Al guardar las cadenas `'a'` , `'b'` , `'b'` , `'c'` en una estructura de datos establecida, hemos perdido la información sobre el hecho de que `'b'` ocurre dos veces. Por supuesto, guardar los elementos en una lista conservaría esta información

```
>>> listA = ['a','b','b','c']
>>> listA
['a', 'b', 'b', 'c']
```

pero una lista de estructura de datos introduce un orden adicional innecesario que ralentizará nuestros cálculos.

Para implementar multisets, Python proporciona la clase `Counter` desde el módulo de `collections`

(a partir de la versión 2.7):

Python 2.x 2.7

```
>>> from collections import Counter
>>> counterA = Counter(['a','b','b','c'])
>>> counterA
Counter({'b': 2, 'a': 1, 'c': 1})
```

`Counter` es un diccionario donde los elementos se almacenan como claves de diccionario y sus conteos se almacenan como valores de diccionario. Y como todos los diccionarios, es una colección desordenada.

Establecer operaciones usando métodos e incorporaciones

Definimos dos conjuntos `a` y `b`

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

> NOTA: `{1}` crea un conjunto de un elemento, pero `{}` crea un `dict` vacío. La forma correcta de crear un conjunto vacío es `set()` .

Intersección

`a.intersection(b)` devuelve un nuevo conjunto con elementos presentes tanto en `a` como en `b`

```
>>> a.intersection(b)
{3, 4}
```

Unión

`a.union(b)` devuelve un nuevo conjunto con elementos presentes en `a` y `b`

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

Diferencia

`a.difference(b)` devuelve un nuevo conjunto con elementos presentes en `a` pero no en `b`

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

Diferencia simétrica

`a.symmetric_difference(b)` devuelve un nuevo conjunto con elementos presentes en ya sea `a` o `b` pero no en ambas

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

NOTA : `a.symmetric_difference(b) == b.symmetric_difference(a)`

Subconjunto y superconjunto

`c.issubset(a)` comprueba si cada elemento de `c` está en `a` .

`a.issuperset(c)` comprueba si cada elemento de `c` está en `a` .

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

Las últimas operaciones tienen operadores equivalentes como se muestra a continuación:

Método	Operador
`a.intersection(b)`	`a & b`
`a.union(b)`	`a | b`
`a.difference(b)`	`a - b`
`a.symmetric_difference(b)`	`a ^ b`
`a.issubset(b)`	`a <= b`
`a.issuperset(b)`	`a >= b`

Conjuntos desunidos

Los conjuntos `a` y `d` son disjuntos si ningún elemento en `a` también está en `d` y viceversa.

```
>>> d = {5, 6}
>>> a.isdisjoint(b) # {2, 3, 4} are in both sets
False
```

```
>>> a.isdisjoint(d)
True

# This is an equivalent check, but less efficient
>>> len(a & d) == 0
True

# This is even less efficient
>>> a & d == set()
True
```

Membresía de prueba

El incorporado `in` palabra clave busca ocurrencias

```
>>> 1 in a
True
>>> 6 in a
False
```

Longitud

La función `len()` incorporada devuelve el número de elementos en el conjunto

```
>>> len(a)
4
>>> len(b)
3
```

Conjunto de conjuntos

```
{{1,2}, {3,4}}
```

lleva a:

```
TypeError: unhashable type: 'set'
```

En su lugar, utilice `frozenset` :

```
{frozenset({1, 2}), frozenset({3, 4})}
```

Lea Conjunto en línea: https://riptutorial.com/es/python/topic/497/conjunto

Capítulo 41: Contando

Examples

Contando todas las apariciones de todos los elementos en un iterable: colecciones.Contador

```
from collections import Counter

c = Counter(["a", "b", "c", "d", "a", "b", "a", "c", "d"])
c
# Out: Counter({'a': 3, 'b': 2, 'c': 2, 'd': 2})
c["a"]
# Out: 3

c[7]     # not in the list (7 occurred 0 times!)
# Out: 0
```

Las `collections.Counter` se puede utilizar para cualquier iterable y cuenta cada aparición para cada elemento.

Nota : una excepción es si se otorga un `dict` u otra `collections.Mapping` crear una clase similar a la de un `dict` , no se contabilizarán, sino que creará un contador con estos valores:

```
Counter({"e": 2})
# Out: Counter({"e": 2})

Counter({"e": "e"})        # warning Counter does not verify the values are int
# Out: Counter({"e": "e"})
```

Obtención del valor más común (-s): collections.Counter.most_common ()

No es posible contar las *claves* de un `Mapping` con `collections.Counter` Contador, pero podemos contar los *valores* :

```
from collections import Counter
adict = {'a': 5, 'b': 3, 'c': 5, 'd': 2, 'e':2, 'q': 5}
Counter(adict.values())
# Out: Counter({2: 2, 3: 1, 5: 3})
```

Los elementos más comunes están disponibles por el método `most_common` :

```
# Sorting them from most-common to least-common value:
Counter(adict.values()).most_common()
# Out: [(5, 3), (2, 2), (3, 1)]

# Getting the most common value
Counter(adict.values()).most_common(1)
# Out: [(5, 3)]
```

```
# Getting the two most common values
Counter(adict.values()).most_common(2)
# Out: [(5, 3), (2, 2)]
```

Contando las ocurrencias de un elemento en una secuencia: list.count () y tuple.count ()

```
alist = [1, 2, 3, 4, 1, 2, 1, 3, 4]
alist.count(1)
# Out: 3

atuple = ('bear', 'weasel', 'bear', 'frog')
atuple.count('bear')
# Out: 2
atuple.count('fox')
# Out: 0
```

Contando las ocurrencias de una subcadena en una cadena: str.count ()

```
astring = 'thisisashorttext'
astring.count('t')
# Out: 4
```

Esto funciona incluso para subcadenas de más de un carácter:

```
astring.count('th')
# Out: 1
astring.count('is')
# Out: 2
astring.count('text')
# Out: 1
```

lo que no sería posible con `collections.Counter` que solo cuenta con caracteres individuales:

```
from collections import Counter
Counter(astring)
# Out: Counter({'a': 1, 'e': 1, 'h': 2, 'i': 2, 'o': 1, 'r': 1, 's': 3, 't': 4, 'x': 1})
```

Contando ocurrencias en matriz numpy

Para contar las ocurrencias de un valor en una matriz numpy. Esto funcionará:

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

La lógica es que la declaración booleana produce una matriz donde todas las apariciones de los valores solicitados son 1 y todas las demás son cero. Así que sumando estos da el número de ocurrencias. Esto funciona para matrices de cualquier forma o tipo de dtype.

Hay dos métodos que utilizo para contar las ocurrencias de todos los valores únicos en numpy. Único y bincount. Unique automáticamente aplana las matrices multidimensionales, mientras que bincount solo funciona con matrices 1d que solo contienen enteros positivos.

```
>>> unique,counts=np.unique(a,return_counts=True)
>>> print unique,counts # counts[i] is equal to occurrences of unique[i] in a
[0 3 4 5 7] [1 2 2 1 1]
>>> bin_count=np.bincount(a)
>>> print bin_count # bin_count[i] is equal to occurrences of i in a
[1 0 0 2 2 1 0 1]
```

Si sus datos son matrices numpy, generalmente es mucho más rápido usar métodos numpy que convertir sus datos a métodos genéricos.

Lea Contando en línea: https://riptutorial.com/es/python/topic/476/contando

Capítulo 42: Copiando datos

Examples

Realizando una copia superficial

Una copia superficial es una copia de una colección sin realizar una copia de sus elementos.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

Realizando una copia profunda

Si tiene listas anidadas, también es deseable clonar las listas anidadas. Esta acción se llama copia profunda.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

Realizando una copia superficial de una lista

Puedes crear copias superficiales de listas usando cortes.

```
>>> l1 = [1,2,3]
>>> l2 = l1[:]     # Perform the shallow copy.
>>> l2
[1,2,3]
>>> l1 is l2
False
```

Copiar un diccionario

Un objeto de diccionario tiene el método de `copy` . Realiza una copia superficial del diccionario.

```
>>> d1 = {1:[]}
>>> d2 = d1.copy()
>>> d1 is d2
False
>>> d1[1] is d2[1]
```

```
True
```

Copiar un conjunto

Los conjuntos también tienen un método de `copy` . Puede utilizar este método para realizar una copia superficial.

```
>>> s1 = {()}
>>> s2 = s1.copy()
>>> s1 is s2
False
>>> s2.add(3)
>>> s1
{[]}
>>> s2
{3,[]}
```

Lea Copiando datos en línea: https://riptutorial.com/es/python/topic/920/copiando-datos

Capítulo 43: Corte de listas (selección de partes de listas)

Sintaxis

- a [inicio: final] # los elementos comienzan hasta el final-1
- a [inicio:] # los elementos comienzan en el resto de la matriz
- a [: fin] # elementos desde el principio hasta el final-1
- a [inicio: final: paso] # comienzo a través de no pasado, paso a paso
- a [:] # una copia de toda la matriz
- fuente

Observaciones

- `lst[::-1]` le da una copia invertida de la lista
- `start` o el `end` puede ser un número negativo, lo que significa que cuenta desde el final de la matriz en lugar del principio. Asi que:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

(fuente)

Examples

Usando el tercer argumento del "paso"

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

Seleccionando una lista secundaria de una lista

```
lst = ['a', 'b', 'c', 'd', 'e']

lst[2:4]
# Output: ['c', 'd']

lst[2:]
# Output: ['c', 'd', 'e']
```

```
lst[:4]
# Output: ['a', 'b', 'c', 'd']
```

Invertir una lista con rebanar

```
a = [1, 2, 3, 4, 5]

# steps through the list backwards (step=-1)
b = a[::-1]

# built-in list method to reverse 'a'
a.reverse()

if a = b:
    print(True)

print(b)

# Output:
# True
# [5, 4, 3, 2, 1]
```

Desplazando una lista usando rebanar

```
def shift_list(array, s):
    """Shifts the elements of a list to the left or right.

    Args:
        array - the list to shift
        s - the amount to shift the list ('+': right-shift, '-': left-shift)

    Returns:
        shifted_array - the shifted list
    """
    # calculate actual shift amount (e.g., 11 --> 1 if length of the array is 5)
    s %= len(array)

    # reverse the shift direction to be more intuitive
    s *= -1

    # shift array with list slicing
    shifted_array = array[s:] + array[:s]

    return shifted_array

my_array = [1, 2, 3, 4, 5]

# negative numbers
shift_list(my_array, -7)
>>> [3, 4, 5, 1, 2]

# no shift on numbers equal to the size of the array
shift_list(my_array, 5)
>>> [1, 2, 3, 4, 5]

# works on positive numbers
shift_list(my_array, 3)
```

```
>>> [3, 4, 5, 1, 2]
```

Lea Corte de listas (selección de partes de listas) en línea:
https://riptutorial.com/es/python/topic/1494/corte-de-listas--seleccion-de-partes-de-listas-

Capítulo 44: Creando paquetes de Python

Observaciones

El proyecto de ejemplo pypa contiene una plantilla `setup.py` fácilmente modificable y completa que demuestra una amplia gama de capacidades que las herramientas de configuración pueden ofrecer.

Examples

Introducción

Cada paquete requiere un archivo `setup.py` que describe el paquete.

Considere la siguiente estructura de directorio para un paquete simple:

```
+-- package_name
|       |
|       +-- __init__.py
|
+-- setup.py
```

El `__init__.py` contiene solo la línea `def foo(): return 100` .

El siguiente `setup.py` definirá el paquete:

```
from setuptools import setup


setup(
    name='package_name',                    # package name
    version='0.1',                          # version
    description='Package Description',      # short description
    url='http://example.com',               # package URL
    install_requires=[],                    # list of packages this package depends
                                            # on.
    packages=['package_name'],              # List of module names that installing
                                            # this package will provide.
)
```

virtualenv es ideal para probar las instalaciones de paquetes sin modificar sus otros entornos de Python:

```
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
$ python setup.py install
running install
...
Installed .../package_name-0.1-....egg
```

```
...
$ python
>>> import package_name
>>> package_name.foo()
100
```

Subiendo a PyPI

Una vez que su `setup.py` sea completamente funcional (vea Introducción), es muy fácil cargar su paquete a PyPI .

Configurar un archivo .pypirc

Este archivo almacena inicios de sesión y contraseñas para autenticar sus cuentas. Normalmente se almacena en su directorio personal.

```
# .pypirc file

[distutils]
index-servers =
  pypi
  pypitest

[pypi]
repository=https://pypi.python.org/pypi
username=your_username
password=your_password

[pypitest]
repository=https://testpypi.python.org/pypi
username=your_username
password=your_password
```

Es más seguro usar `twine` para cargar paquetes, así que asegúrese de que esté instalado.

```
$ pip install twine
```

Registrarse y subir a testpypi (opcional)

Nota : PyPI no permite sobrescribir paquetes cargados , por lo que es prudente probar primero su implementación en un servidor de prueba dedicado, por ejemplo, testpypi. Esta opción será discutida. Considere un esquema de control de versiones para su paquete antes de cargar, como el control de versiones del calendario o el control de versiones semántico .

Inicie sesión o cree una nueva cuenta en testpypi . El registro solo se requiere la primera vez, aunque registrarse más de una vez no es perjudicial.

```
$ python setup.py register -r pypitest
```

Mientras que en el directorio raíz de su paquete:

```
$ twine upload dist/* -r pypitest
```

Su paquete ahora debe ser accesible a través de su cuenta.

Pruebas

Realiza un entorno virtual de prueba. Intente `pip install` su paquete desde testpypi o PyPI.

```
# Using virtualenv
$ mkdir testenv
$ cd testenv
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
# Test from testpypi
(.virtualenv)  pip install --verbose --extra-index-url https://testpypi.python.org/pypi
package_name
...
# Or test from PyPI
(.virtualenv) $ pip install package_name
...

(.virtualenv) $ python
Python 3.5.1 (default, Jan 27 2016, 19:16:39)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import package_name
>>> package_name.foo()
100
```

Si tiene éxito, su paquete es menos importable. Puede considerar probar su API también antes de su carga final a PyPI. Si el paquete falló durante la prueba, no se preocupe. Aún puedes arreglarlo, volver a subirlo a testpypi y probar de nuevo.

Registrarse y subir a PyPI

Asegúrate de que el `twine` esté instalado:

```
$ pip install twine
```

Inicie sesión o cree una nueva cuenta en PyPI .

```
$ python setup.py register -r pypi
$ twine upload dist/*
```

¡Eso es! Su paquete ya está en vivo .

Si descubre un error, simplemente cargue una nueva versión de su paquete.

Documentación

No olvide incluir al menos algún tipo de documentación para su paquete. PyPi toma como idioma de formato predeterminado reStructuredText .

Readme

Si su paquete no tiene una gran documentación, incluya lo que puede ayudar a otros usuarios en el archivo `README.rst` . Cuando el archivo está listo, se necesita otro para decirle a PyPi que lo muestre.

Cree el archivo `setup.cfg` y ponga estas dos líneas en él:

```
[metadata]
description-file = README.rst
```

Tenga en cuenta que si intenta colocar el archivo Markdown en su paquete, PyPi lo leerá como un archivo de texto puro sin ningún formato.

Licenciamiento

A menudo es más que bienvenido poner un archivo `LICENSE.txt` en su paquete con una de las licencias de OpenSource para informar a los usuarios si pueden usar su paquete, por ejemplo, en proyectos comerciales o si su código se puede usar con su licencia.

De manera más legible, algunas licencias se explican en TL; DR .

Haciendo paquete ejecutable

Si su paquete no es solo una biblioteca, sino que tiene una pieza de código que puede usarse como una vitrina o una aplicación independiente cuando su paquete está instalado, coloque esa pieza de código en el archivo `__main__.py` .

Ponga la `__main__.py` en el `package_name` carpeta. De esta manera podrás ejecutarlo directamente desde la consola:

```
python -m package_name
```

Si no hay `__main__.py` archivo `__main__.py` disponible, el paquete no se ejecutará con este comando y se imprimirá este error:

> python: No hay un módulo llamado `package_name.__main__`; 'package_name' es un paquete y no se puede ejecutar directamente.

Lea Creando paquetes de Python en línea: https://riptutorial.com/es/python/topic/1381/creando-

paquetes-de-python

Capítulo 45: Creando un servicio de Windows usando Python

Introducción

Los procesos sin cabeza (sin interfaz de usuario) en Windows se denominan Servicios. Se pueden controlar (iniciar, detener, etc.) mediante los controles estándar de Windows, como la consola de comandos, Powershell o la pestaña Servicios en el Administrador de tareas. Un buen ejemplo podría ser una aplicación que proporcione servicios de red, como una aplicación web, o tal vez una aplicación de respaldo que realice varias tareas de archivado en segundo plano. Hay varias formas de crear e instalar una aplicación de Python como un Servicio en Windows.

Examples

Un script de Python que se puede ejecutar como un servicio

Los módulos utilizados en este ejemplo son parte de pywin32 (Python para extensiones de Windows). Dependiendo de cómo instaló Python, es posible que necesite instalar esto por separado.

```
import win32serviceutil
import win32service
import win32event
import servicemanager
import socket


class AppServerSvc (win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"

    def __init__(self,args):
        win32serviceutil.ServiceFramework.__init__(self,args)
        self.hWaitStop = win32event.CreateEvent(None,0,0,None)
        socket.setdefaulttimeout(60)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

    def SvcDoRun(self):
        servicemanager.LogMsg(servicemanager.EVENTLOG_INFORMATION_TYPE,
                              servicemanager.PYS_SERVICE_STARTED,
                              (self._svc_name_,''))
        self.main()

    def main(self):
        pass

if __name__ == '__main__':
```

```
    win32serviceutil.HandleCommandLine(AppServerSvc)
```

Esto es sólo boilerplate. Su código de aplicación, probablemente invocando un script separado, iría en la función main ().

También necesitarás instalar esto como un servicio. La mejor solución para esto en este momento parece ser utilizar el Administrador de servicios que no chupa . Esto le permite instalar un servicio y proporciona una GUI para configurar la línea de comandos que ejecuta el servicio. Para Python puedes hacer esto, lo que crea el servicio de una sola vez:

```
nssm install MyServiceName c:\python27\python.exe c:\temp\myscript.py
```

Donde my_script.py es el script de boilerplate anterior, modificado para invocar su script o código de aplicación en la función main (). Tenga en cuenta que el servicio no ejecuta la secuencia de comandos de Python directamente, ejecuta el intérprete de Python y le pasa la secuencia de comandos principal en la línea de comandos.

Alternativamente, puede usar las herramientas proporcionadas en el Kit de recursos de Windows Server para la versión de su sistema operativo, así que cree el servicio.

Ejecutando una aplicación web de Flask como un servicio

Esta es una variación del ejemplo genérico. Solo necesita importar el script de su aplicación e invocar el método `run()` en la función `main()` . En este caso, también estamos usando el módulo de multiprocesamiento debido a un problema al acceder a `WSGIRequestHandler` .

```
import win32serviceutil
import win32service
import win32event
import servicemanager
from multiprocessing import Process

from app import app


class Service(win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"
    _svc_description_ = "Tests Python service framework by receiving and echoing messages over
a named pipe"

    def __init__(self, *args):
        super().__init__(*args)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        self.process.terminate()
        self.ReportServiceStatus(win32service.SERVICE_STOPPED)

    def SvcDoRun(self):
        self.process = Process(target=self.main)
        self.process.start()
        self.process.run()
```

```
    def main(self):
        app.run()


if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(Service)
```

Adaptado de http://stackoverflow.com/a/25130524/318488

Lea Creando un servicio de Windows usando Python en línea:
https://riptutorial.com/es/python/topic/9065/creando-un-servicio-de-windows-usando-python

Capítulo 46: Crear entorno virtual con virtualenvwrapper en windows

Examples

Entorno virtual con virtualenvwrapper para windows

Supongamos que necesita trabajar en tres proyectos diferentes. El proyecto A, el proyecto B y el proyecto C. el proyecto A y el proyecto B necesitan Python 3 y algunas bibliotecas requeridas. Pero para el proyecto C necesitas Python 2.7 y bibliotecas dependientes.

Así que la mejor práctica para esto es separar esos entornos de proyecto. Para crear un entorno virtual de Python separado, debe seguir los siguientes pasos:

Paso 1: instala pip con este comando: `python -m pip install -U pip`

Paso 2: Luego instale el paquete "virtualenvwrapper-win" usando el comando (el comando puede ejecutarse en Windows Power Shell):

```
pip install virtualenvwrapper-win
```

Paso 3: Crea un nuevo entorno virtualenv usando el comando: `mkvirtualenv python_3.5`

Paso 4: Activar el entorno mediante el comando:

```
workon < environment name>
```

Comandos principales para virtualenvwrapper:

```
mkvirtualenv <name>
Create a new virtualenv environment named <name>. The environment will be created in
WORKON_HOME.

lsvirtualenv
List all of the enviornments stored in WORKON_HOME.

rmvirtualenv <name>
Remove the environment <name>. Uses folder_delete.bat.

workon [<name>]
If <name> is specified, activate the environment named <name> (change the working virtualenv
to <name>). If a project directory has been defined, we will change into it. If no argument is
specified, list the available environments. One can pass additional option -c after virtualenv
name to cd to virtualenv directory if no projectdir is set.

deactivate
Deactivate the working virtualenv and switch back to the default system Python.

add2virtualenv <full or relative path>
If a virtualenv environment is active, appends <path> to virtualenv_path_extensions.pth inside
```

```
the environment’s site-packages, which effectively adds <path> to the environment’s
PYTHONPATH. If a virtualenv environment is not active, appends <path> to
virtualenv_path_extensions.pth inside the default Python’s site-packages. If <path> doesn’t
exist, it will be created.
```

Lea Crear entorno virtual con virtualenvwrapper en windows en línea: https://riptutorial.com/es/python/topic/9984/crear-entorno-virtual-con-virtualenvwrapper-en-windows

Capítulo 47: ctypes

Introducción

`ctypes` es una biblioteca incorporada de python que invoca funciones exportadas desde bibliotecas compiladas nativas.

Nota: Dado que esta biblioteca maneja el código compilado, es relativamente dependiente del sistema operativo.

Examples

Uso básico

Digamos que queremos usar la `libc ntohl libc` .

Primero, debemos cargar `libc.so` :

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

Entonces, obtenemos el objeto de función:

```
>>> ntohl = libc.ntohl
>>> ntohl
<_FuncPtr object at 0xbaadf00d>
```

Y ahora, simplemente podemos invocar la función:

```
>>> ntohl(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

Lo que hace exactamente lo que esperamos que haga.

Errores comunes

No cargar un archivo

El primer error posible está fallando al cargar la biblioteca. En ese caso, un OSError normalmente se plantea.

Esto se debe a que el archivo no existe (o el sistema operativo no lo puede encontrar):

```
>>> cdll.LoadLibrary("foobar.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: foobar.so: cannot open shared object file: No such file or directory
```

Como puede ver, el error es claro y bastante indicativo.

La segunda razón es que se encuentra el archivo, pero no tiene el formato correcto.

```
>>> cdll.LoadLibrary("libc.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: /usr/lib/i386-linux-gnu/libc.so: invalid ELF header
```

En este caso, el archivo es un archivo de script y no un archivo `.so` . Esto también puede suceder cuando se intenta abrir un archivo `.dll` en una máquina Linux o un archivo de 64 bits en un intérprete de 32 bits de Python. Como puede ver, en este caso el error es un poco más vago y requiere un poco de investigación.

No acceder a una función

Suponiendo que `.so` éxito el archivo `.so` , necesitamos acceder a nuestra función como lo hemos hecho en el primer ejemplo.

Cuando se usa una función que no existe, se `AttributeError` un `AttributeError` :

```
>>> libc.foo
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 360, in __getattr__
    func = self.__getitem__(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 365, in __getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: /lib/i386-linux-gnu/libc.so.6: undefined symbol: foo
```

Objeto de ctypes básico

El objeto más básico es un int:

```
>>> obj = ctypes.c_int(12)
>>> obj
c_long(12)
```

Ahora, `obj` refiere a una porción de memoria que contiene el valor 12.

Se puede acceder a ese valor directamente, e incluso modificarse:

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

Ya que `obj` refiere a un trozo de memoria, también podemos averiguar su tamaño y ubicación:

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

arrays de ctypes

Como cualquier buen programador de C sabe, un solo valor no lo llevará tan lejos. ¡Lo que realmente nos hará avanzar son matrices!

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

Esto no es una matriz real, pero está bastante cerca! Creamos una clase que denota una matriz de 16 `int` s.

Ahora todo lo que tenemos que hacer es inicializarlo:

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcafe>
```

Ahora `arr` es una matriz real que contiene los números del 0 al 15.

Se puede acceder a ellos como en cualquier lista:

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

Y al igual que cualquier otro objeto `ctypes` , también tiene un tamaño y una ubicación:

```
>>> sizeof(arr)
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc000l0ff'
```

Funciones de envoltura para ctypes.

En algunos casos, una función C acepta un puntero de función. Como usuarios ávidos de `ctypes` , nos gustaría usar esas funciones, e incluso pasar la función python como argumentos.

Vamos a definir una función:

```
>>> def max(x, y):
        return x if x >= y else y
```

Ahora, esa función toma dos argumentos y devuelve un resultado del mismo tipo. Por el bien del ejemplo, asumamos que el tipo es un int.

Como hicimos en el ejemplo de matriz, podemos definir un objeto que denota ese prototipo:

```
>>> CFUNCTYPE(c_int, c_int, c_int)
<CFunctionType object at 0xdeadbeef>
```

Ese prototipo denota una función que devuelve un `c_int` (el primer argumento), y acepta dos argumentos `c_int` (los otros argumentos).

Ahora vamos a envolver la función:

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

Los prototipos de funciones tienen un mayor uso: pueden envolver la función `ctypes` (como `libc.ntohl`) y verificar que se usan los argumentos correctos cuando se invoca la función.

```
>>> libc.ntohl() # garbage in - garbage out
>>> CFUNCTYPE(c_int, c_int)(libc.ntohl)()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: this function takes at least 1 argument (0 given)
```

Uso complejo

`lfind` todos los ejemplos anteriores en un escenario complejo: utilizando la `libc` `lfind` `libc` .

Para más detalles sobre la función, lea la página del manual . Les insto a que lo lean antes de continuar.

Primero, definiremos los prototipos adecuados:

```
>>> compar_proto = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> lfind_proto = CFUNCTYPE(c_void_p, c_void_p, c_void_p, POINTER(c_uint), c_uint,
compar_proto)
```

Entonces, vamos a crear las variables:

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

Y ahora definimos la función de comparación:

```
>>> def compar(x, y):
        return x.contents.value - y.contents.value
```

Tenga en cuenta que `x` , y `y` son `POINTER(c_int)` , por lo tanto, debemos desreferenciarlos y tomar sus valores para poder comparar realmente el valor almacenado en la memoria.

Ahora podemos combinar todo juntos:

```
>>> lfind = lfind_proto(libc.lfind)
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```

`ptr` es el puntero vacío devuelto. Si no se encontró la `key` en `arr` , el valor sería `None` , pero en este caso obtuvimos un valor válido.

Ahora podemos convertirlo y acceder al valor:

```
>>> cast(ptr, POINTER(c_int)).contents
c_long(12)
```

También, podemos ver que `ptr` apunta al valor correcto dentro de `arr` :

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr
True
```

Lea ctypes en línea: https://riptutorial.com/es/python/topic/9050/ctypes

Capítulo 48: Datos binarios

Sintaxis

- paquete (fmt, v1, v2, ...)
- desempaquetar (fmt, buffer)

Examples

Formatear una lista de valores en un objeto byte

```
from struct import pack

print(pack('I3c', 123, b'a', b'b', b'c'))  # b'{\x00\x00\x00abc'
```

Desempaquetar un objeto byte de acuerdo con una cadena de formato

```
from struct import unpack

print(unpack('I3c', b'{\x00\x00\x00abc'))  # (123, b'a', b'b', b'c')
```

Embalaje de una estructura

El módulo " **struct** " proporciona facilidad para empaquetar objetos de python como trozos contiguos de bytes o para diseminar un trozo de bytes en estructuras de python.

La función de paquete toma una cadena de formato y uno o más argumentos, y devuelve una cadena binaria. Esto se parece mucho a que está formateando una cadena, excepto que la salida no es una cadena sino una porción de bytes.

```
import struct
import sys
print "Native byteorder: ", sys.byteorder
# If no byteorder is specified, native byteorder is used
buffer = struct.pack("ihb", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
print "Byte chunk unpacked: ", struct.unpack("ihb", buffer)
# Last element as unsigned short instead of unsigned char ( 2 Bytes)
buffer = struct.pack("ihh", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
```

Salida:

> Byteorder nativo: fragmento de bytes pequeño: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x00 \ x00' Fragmento de bytes sin empaquetar: (3, 4, 5) Fragmento de bytes: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x00 \ x05 \ x00 '

Puede usar el orden de bytes de la red con los datos recibidos de la red o datos del paquete para enviarlos a la red.

```
import struct
# If no byteorder is specified, native byteorder is used
buffer = struct.pack("hhh", 3, 4, 5)
print "Byte chunk native byte order: ", repr(buffer)
buffer = struct.pack("!hhh", 3, 4, 5)
print "Byte chunk network byte order: ", repr(buffer)
```

Salida:

> Orden de bytes nativo del byte: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00'
>
> Orden de bytes de la red de bytes: '\ x00 \ x03 \ x00 \ x04 \ x00 \ x05'

Puede optimizar evitando la sobrecarga de asignar un nuevo búfer al proporcionar un búfer que se creó anteriormente.

```
import struct
from ctypes import create_string_buffer
bufferVar = create_string_buffer(8)
bufferVar2 = create_string_buffer(8)
# We use a buffer that has already been created
# provide format, buffer, offset and data
struct.pack_into("hhh", bufferVar, 0, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar.raw)
struct.pack_into("hhh", bufferVar2, 2, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar2.raw)
```

Salida:

> Fragmento de bytes: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00 \ x00 \ x00'
>
> Fragmento de bytes: '\ x00 \ x00 \ x03 \ x00 \ x04 \ x00 \ x05 \ x00'

Lea Datos binarios en línea: https://riptutorial.com/es/python/topic/2978/datos-binarios

Capítulo 49: Decoradores

Introducción

Las funciones de decorador son patrones de diseño de software. Alteran dinámicamente la funcionalidad de una función, método o clase sin tener que usar subclases directamente o cambiar el código fuente de la función decorada. Cuando se usan correctamente, los decoradores pueden convertirse en herramientas poderosas en el proceso de desarrollo. Este tema cubre la implementación y las aplicaciones de las funciones de decorador en Python.

Sintaxis

- def decorator_function (f): pass # define un decorador llamado decorator_function
- @decorator_function
  def decorated_function (): pass # la función ahora está envuelta (decorada por) decorator_function
- decorated_function = decorator_function (decorated_function) # esto es equivalente a usar el azúcar sintáctico `@decorator_function`

Parámetros

Parámetro	Detalles
F	La función a decorar (envolver)

Examples

Función decoradora

Los decoradores aumentan el comportamiento de otras funciones o métodos. Cualquier función que tome una función como parámetro y devuelva una función aumentada puede usarse como **decorador** .

```
# This simplest decorator does nothing to the function being decorated. Such
# minimal decorators can occasionally be used as a kind of code markers.
def super_secret_function(f):
    return f

@super_secret_function
def my_function():
    print("This is my secret function.")
```

La notación `@` es azúcar sintáctica que es equivalente a lo siguiente:

```
my_function = super_secret_function(my_function)
```

Es importante tener esto en cuenta para comprender cómo funcionan los decoradores. Esta sintaxis "no saturada" deja claro por qué la función decoradora toma una función como argumento y por qué debería devolver otra función. También demuestra lo que sucedería si *no* devuelves una función:

```
def disabled(f):
    """
    This function returns nothing, and hence removes the decorated function
    from the local scope.
    """
    pass

@disabled
def my_function():
    print("This function can no longer be called...")

my_function()
# TypeError: 'NoneType' object is not callable
```

Por lo tanto, generalmente definimos una *nueva función* dentro del decorador y la devolvemos. Esta nueva función primero haría algo que debe hacer, luego llama a la función original y, finalmente, procesa el valor de retorno. Considere esta función decoradora simple que imprime los argumentos que recibe la función original y luego la llama.

```
#This is the decorator
def print_args(func):
    def inner_func(*args, **kwargs):
        print(args)
        print(kwargs)
        return func(*args, **kwargs) #Call the original function with its arguments.
    return inner_func

@print_args
def multiply(num_a, num_b):
    return num_a * num_b

print(multiply(3, 5))
#Output:
# (3,5) - This is actually the 'args' that the function receives.
# {} - This is the 'kwargs', empty because we didn't specify keyword arguments.
# 15 - The result of the function.
```

Clase de decorador

Como se mencionó en la introducción, un decorador es una función que se puede aplicar a otra función para aumentar su comportamiento. El azúcar sintáctico es equivalente a lo siguiente: `my_func = decorator(my_func)` . Pero, ¿y si el `decorator` fuera una clase? La sintaxis aún funcionaría, excepto que ahora `my_func` se reemplaza con una instancia de la clase `decorator` . Si esta clase implementa el método mágico `__call__()` , entonces todavía sería posible usar `my_func` como si fuera una función:

```
class Decorator(object):
    """Simple decorator class."""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Before the function call.')
        res = self.func(*args, **kwargs)
        print('After the function call.')
        return res

@Decorator
def testfunc():
    print('Inside the function.')

testfunc()
# Before the function call.
# Inside the function.
# After the function call.
```

Tenga en cuenta que una función decorada con un decorador de clase ya no se considerará una "función" desde la perspectiva de la comprobación de tipo:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Métodos de decoración

Para los métodos de decoración debe definir un método `__get__` adicional:

```
from types import MethodType

class Decorator(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Inside the decorator.')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        # Return a Method if it is called on an instance
        return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass

a = Test()
```

Dentro del decorador.

¡Advertencia!

Los decoradores de clase solo producen una instancia para una función específica, por lo que decorar un método con un decorador de clase compartirá el mismo decorador entre todas las instancias de esa clase:

```
from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0    # Number of calls of this method

    def __call__(self, *args, **kwargs):
        self.ncalls += 1   # Increment the calls counter
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class Test(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return 'something was done'

a = Test()
a.do_something()
a.do_something.ncalls   # 1
b = Test()
b.do_something()
b.do_something.ncalls   # 2
```

Hacer que un decorador se vea como la función decorada.

Los decoradores normalmente eliminan los metadatos de la función ya que no son lo mismo. Esto puede causar problemas cuando se utiliza la meta-programación para acceder dinámicamente a los metadatos de la función. Los metadatos también incluyen las cadenas de documentación de la función y su nombre. `functools.wraps` hace que la función decorada se vea como la función original al copiar varios atributos a la función de envoltura.

```
from functools import wraps
```

Los dos métodos de envolver a un decorador están logrando lo mismo al ocultar que la función original ha sido decorada. No hay razón para preferir la versión de la función a la versión de clase a menos que ya esté utilizando una sobre la otra.

Como una función

```
def decorator(func):
    # Copies the docstring, name, annotations and module to the decorator
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func

@decorator
def test():
    pass

test.__name__
```

'prueba'

Como una clase

```
class Decorator(object):
    def __init__(self, func):
        # Copies name, module, annotations and docstring to the instance.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
```

'Docstring of test'.

Decorador con argumentos (decorador de fábrica).

Un decorador toma solo un argumento: la función a decorar. No hay forma de pasar otros argumentos.

Pero a menudo se desean argumentos adicionales. El truco es, entonces, hacer una función que tome argumentos arbitrarios y devuelva un decorador.

Funciones de decorador

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
```

```
            print('The decorator wants to tell you: {}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@decoratorfactory('Hello World')
def test():
    pass

test()
```

El decorador quiere decirte: Hola Mundo.

Nota IMPORTANTE:

Con tales fábricas de decoradores **debe** llamar al decorador con un par de paréntesis:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

TypeError: decorator () falta 1 argumento posicional requerido: 'func'

Clases de decorador

```
def decoratorfactory(*decorator_args, **decorator_kwargs):

    class Decorator(object):
        def __init__(self, func):
            self.func = func

        def __call__(self, *args, **kwargs):
            print('Inside the decorator with arguments {}'.format(decorator_args))
            return self.func(*args, **kwargs)

    return Decorator

@decoratorfactory(10)
def test():
    pass

test()
```

Dentro del decorador con argumentos (10,)

Crea una clase de singleton con un decorador.

Un singleton es un patrón que restringe la creación de instancias de una clase a una instancia / objeto. Usando un decorador, podemos definir una clase como un singleton forzando a la clase a

devolver una instancia existente de la clase o crear una nueva instancia (si no existe).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

Este decorador se puede agregar a cualquier declaración de clase y se asegurará de que se cree como máximo una instancia de la clase. Cualquier llamada posterior devolverá la instancia de clase ya existente.

```
@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Created!")

instance = SomeSingletonClass()  # prints: Created!
instance = SomeSingletonClass()  # doesn't print anything
print(instance.x)                # 2

instance.x = 3
print(SomeSingletonClass().x)    # 3
```

Por lo tanto, no importa si se refiere a la instancia de clase a través de su variable local o si crea otra "instancia", siempre obtiene el mismo objeto.

Usando un decorador para cronometrar una función.

```
import time
def timer(func):
    def inner(*args, **kwargs):
        t1 = time.time()
        f = func(*args, **kwargs)
        t2 = time.time()
        print 'Runtime took {0} seconds'.format(t2-t1)
        return f
    return inner

@timer
def example_function():
    #do stuff


example_function()
```

Lea Decoradores en línea: https://riptutorial.com/es/python/topic/229/decoradores

Capítulo 50: Definiendo funciones con argumentos de lista

Examples

Función y Llamada

Las listas como argumentos son solo otra variable:

```
def func(myList):
    for item in myList:
        print(item)
```

y se puede pasar en la llamada de función en sí:

```
func([1,2,3,5,7])

1
2
3
5
7
```

O como una variable:

```
aList = ['a','b','c','d']
func(aList)

a
b
c
d
```

Lea Definiendo funciones con argumentos de lista en línea:
https://riptutorial.com/es/python/topic/7744/definiendo-funciones-con-argumentos-de-lista

Capítulo 51: dejar de lado

Introducción

Shelve es un módulo de Python que se utiliza para almacenar objetos en un archivo. El módulo de almacenamiento implementa el almacenamiento persistente para objetos Python arbitrarios que pueden ser decapados, utilizando una API similar a un diccionario. El módulo de almacenamiento puede usarse como una opción de almacenamiento persistente simple para objetos de Python cuando una base de datos relacional es excesiva. Se accede a la estantería mediante llaves, igual que con un diccionario. Los valores se decapan y se escriben en una base de datos creada y administrada por anydbm.

Observaciones

Nota: No confíe en que el estante se cierre automáticamente; siempre llame a `close()` explícitamente cuando ya no lo necesite más, o use `shelve.open()` como administrador de contexto:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Advertencia:

Debido a que la `shelve` módulo está respaldado por `pickle` , es inseguro para cargar un estante de una fuente no fiable. Al igual que con Pickle, cargar un estante puede ejecutar código arbitrario.

Restricciones

1 . La elección del paquete de base de datos que se utilizará (como dbm.ndbm o dbm.gnu) depende de la interfaz disponible. Por lo tanto, no es seguro abrir la base de datos directamente usando dbm. La base de datos también está (desafortunadamente) sujeta a las limitaciones de dbm, si se usa; esto significa que (la representación en escabeche) de los objetos almacenados en la base de datos debería ser bastante pequeña, y en casos raros, las colisiones de claves pueden hacer que la base de datos rechazar actualizaciones.

2. El módulo de archivado no admite el acceso simultáneo de lectura / escritura a objetos archivados. (Múltiples accesos de lectura simultáneos son seguros.) Cuando un programa tiene un estante abierto para escribir, ningún otro programa debería tenerlo abierto para leer o escribir. El bloqueo de archivos de Unix se puede usar para resolver esto, pero esto difiere entre las versiones de Unix y requiere conocimiento sobre la implementación de la base de datos utilizada.

Examples

Código de muestra para estantería

Para archivar un objeto, primero importe el módulo y luego asigne el valor del objeto de la siguiente manera:

```
import shelve
 database = shelve.open(filename.suffix)
 object = Object()
 database['key'] = object
```

Para resumir la interfaz (la clave es una cadena, los datos son un objeto arbitrario):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

Creando un nuevo estante

La forma más sencilla de usar shelve es a través de la clase **DbfilenameShelf** . Utiliza anydbm para almacenar los datos. Puede usar la clase directamente, o simplemente llamar a **shelve.open ()** :

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
```

```
finally:
    s.close()
```

Para volver a acceder a los datos, abra el estante y utilícelo como un diccionario:

```
    import shelve

    s = shelve.open('test_shelf.db')
    try:
        existing = s['key1']
    finally:
        s.close()

print existing
```

Si ejecuta ambos scripts de muestra, debería ver:

```
$ python shelve_create.py
$ python shelve_existing.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

El módulo **dbm** no admite múltiples aplicaciones que escriban en la misma base de datos al mismo tiempo. Si sabe que su cliente no modificará el estante, puede pedirle a la estantería que abra la base de datos de solo lectura.

```
import shelve

s = shelve.open('test_shelf.db', flag='r')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

Si su programa intenta modificar la base de datos mientras se abre solo para lectura, se genera una excepción de error de acceso. El tipo de excepción depende del módulo de base de datos seleccionado por anydbm cuando se creó la base de datos.

Respóndeme

Los estantes no rastrean las modificaciones a objetos volátiles, por defecto. Eso significa que si cambia el contenido de un elemento almacenado en el estante, debe actualizar el estante explícitamente almacenándolo nuevamente.

```
import shelve

s = shelve.open('test_shelf.db')
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
finally:
```

```
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

En este ejemplo, el diccionario en 'key1' no se almacena de nuevo, por lo que cuando el estante se vuelve a abrir, los cambios no se han conservado.

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

Para capturar automáticamente los cambios en los objetos volátiles almacenados en el estante, abra el estante con la función de escritura habilitada. El indicador de reescritura hace que el estante recuerde todos los objetos recuperados de la base de datos utilizando un caché en memoria. Cada objeto de caché también se vuelve a escribir en la base de datos cuando se cierra el estante.

```
import shelve

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
    print s['key1']
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

Aunque reduce la posibilidad de error del programador y puede hacer que la persistencia del objeto sea más transparente, el uso del modo de reescritura puede no ser deseable en todas las situaciones. La memoria caché consume memoria adicional mientras el estante está abierto, y hacer una pausa para escribir cada objeto almacenado en caché de nuevo en la base de datos cuando se cierra puede llevar más tiempo. Dado que no hay forma de saber si los objetos almacenados en caché se han modificado, todos se han vuelto a escribir. Si su aplicación lee datos más de lo que escribe, la respuesta por escrito agregará más sobrecarga de la que podría desear.

```
$ python shelve_create.py
$ python shelve_writeback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
```

```
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
```

Lea dejar de lado en línea: https://riptutorial.com/es/python/topic/10629/dejar-de-lado

Capítulo 52: Depuración

Examples

El depurador de Python: depuración paso a paso con _pdb_

La biblioteca estándar de Python incluye una biblioteca de depuración interactiva llamada *pdb* . *pdb* tiene capacidades extensivas, la más comúnmente utilizada es la capacidad de "avanzar" en un programa.

Para entrar de inmediato en el uso de depuración paso a paso:

```
python -m pdb <my_file.py>
```

Esto iniciará el depurador en la primera línea del programa.

Por lo general, deseará apuntar a una sección específica del código para la depuración. Para hacer esto, importamos la biblioteca pdb y usamos *set_trace ()* para interrumpir el flujo de este código de ejemplo problemático.

```
import pdb

def divide(a, b):
    pdb.set_trace()
    return a/b
    # What's wrong with this? Hint: 2 != 3

print divide(1, 2)
```

Ejecutando este programa se iniciará el depurador interactivo.

```
python foo.py
> ~/scratch/foo.py(5)divide()
-> return a/b
(Pdb)
```

A menudo, este comando se usa en una línea, por lo que se puede comentar con un solo # carácter

```
 import pdf; pdb.set_trace()
```

En el *comando (Pdb)* se pueden introducir comandos. Estos comandos pueden ser comandos de depuración o python. Para imprimir variables podemos usar *p* del depurador, o la *impresión* de python.

```
(Pdb) p a
1
(Pdb) print a
```

```
1
```

Para ver la lista de todas las variables locales use

```
locals
```

función incorporada

Estos son buenos comandos de depuración para saber:

```
b <n> | <f>: set breakpoint at line *n* or function named *f*.
# b 3
# b divide
b: show all breakpoints.
c: continue until the next breakpoint.
s: step through this line (will enter a function).
n: step over this line (jumps over a function).
r: continue until the current function returns.
l: list a window of code around this line.
p <var>: print variable named *var*.
# p x
q: quit debugger.
bt: print the traceback of the current execution call stack
up: move your scope up the function call stack to the caller of the current function
down: Move your scope back down the function call stack one level
step: Run the program until the next line of execution in the program, then return control
back to the debugger
next: run the program until the next line of execution in the current function, then return
control back to the debugger
return: run the program until the current function returns, then return control back to the
debugger
continue: continue running the program until the next breakpoint (or set_trace si called
again)
```

El depurador también puede evaluar python interactivamente:

```
-> return a/b
(Pdb) p a+b
3
(Pdb) [ str(m) for m in [a,b]]
['1', '2']
(Pdb) [ d for d in xrange(5)]
[0, 1, 2, 3, 4]
```

Nota:

Si alguno de sus nombres de variable coincide con los comandos del depurador, use un signo de exclamación ' **!** 'antes de la var para referirse explícitamente a la variable y no al comando del depurador. Por ejemplo, a menudo puede suceder que use el nombre de variable ' **c** ' para un contador, y tal vez desee imprimirlo mientras está en el depurador. un simple comando ' **c** ' continuaría la ejecución hasta el siguiente punto de interrupción. En su lugar, use ' **! C** ' para imprimir el valor de la variable de la siguiente manera:

```
(Pdb) !c
```

```
4
```

A través de IPython y ipdb

Si se instala IPython (o Jupyter), se puede invocar al depurador usando:

```
import ipdb
ipdb.set_trace()
```

Cuando se alcanza, el código saldrá e imprimirá:

```
 /home/usr/ook.py(3)<module>()
      1 import ipdb
      2 ipdb.set_trace()
----> 3 print("Hello world!")

ipdb>
```

Claramente, esto significa que uno tiene que editar el código. Hay una forma más sencilla:

```
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
                                     color_scheme='Linux',
                                     call_pdb=1)
```

Esto hará que se llame al depurador si se produce una excepción no detectada.

Depurador remoto

Algunas veces necesitas depurar el código python que se ejecuta mediante otro proceso y, en este caso, `rpdb` es muy útil.

> rpdb es un envoltorio alrededor de pdb que redirige stdin y stdout a un controlador de socket. Por defecto abre el depurador en el puerto 4444.

Uso:

```
# In the Python file you want to debug.
import rpdb
rpdb.set_trace()
```

Y luego necesitas ejecutar esto en la terminal para conectarte a este proceso.

```
# Call in a terminal to see the output
$ nc 127.0.0.1 4444
```

Y obtendrás pdb promt

```
> /home/usr/ook.py(3)<module>()
-> print("Hello world!")
```

```
(Pdb)
```

Lea Depuración en línea: https://riptutorial.com/es/python/topic/2077/depuracion

Capítulo 53: Descomprimir archivos

Introducción

Para extraer o descomprimir un archivo tar, ZIP o gzip, se proporcionan los módulos tarfile, zipfile y gzip de Python, respectivamente. El módulo `TarFile.extractall(path=".", members=None)` Python proporciona la función `TarFile.extractall(path=".", members=None)` para extraer de un archivo tarball. El módulo zipfile de Python proporciona la función `ZipFile.extractall([path[, members[, pwd]]])` para extraer o descomprimir archivos comprimidos ZIP. Finalmente, el módulo gzip de Python proporciona la clase GzipFile para descomprimir.

Examples

Usando Python ZipFile.extractall () para descomprimir un archivo ZIP

```
file_unzip = 'filename.zip'
unzip = zipfile.ZipFile(file_unzip, 'r')
unzip.extractall()
unzip.close()
```

Usando Python TarFile.extractall () para descomprimir un tarball

```
file_untar = 'filename.tar.gz'
untar = tarfile.TarFile(file_untar)
untar.extractall()
untar.close()
```

Lea Descomprimir archivos en línea: https://riptutorial.com/es/python/topic/9505/descomprimir-archivos

Capítulo 54: Descriptor

Examples

Descriptor simple

Hay dos tipos diferentes de descriptores. Los descriptores de datos se definen como objetos que definen tanto un `__get__()` como un `__set__()` , mientras que los descriptores que no son de datos solo definen un `__get__()` . Esta distinción es importante cuando se consideran las sustituciones y el espacio de nombres del diccionario de una instancia. Si un descriptor de datos y una entrada en el diccionario de una instancia comparten el mismo nombre, el descriptor de datos tendrá prioridad. Sin embargo, si en cambio un descriptor que no es de datos y una entrada en el diccionario de una instancia comparten el mismo nombre, la entrada del diccionario de la instancia tendrá prioridad.

Para hacer un descriptor de datos de solo lectura, defina tanto **get** () como **set** () con el **conjunto** () generando un AttributeError cuando se le llama. Definir el método **set** () con un marcador de posición de aumento de excepción es suficiente para convertirlo en un descriptor de datos.

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Un ejemplo implementado:

```
class DescPrinter(object):
    """A data descriptor that logs activity."""
    _val = 7

    def __get__(self, obj, objtype=None):
        print('Getting ...')
        return self._val

    def __set__(self, obj, val):
        print('Setting', val)
        self._val = val

    def __delete__(self, obj):
        print('Deleting ...')
        del self._val


class Foo():
    x = DescPrinter()

i = Foo()
i.x
# Getting ...
# 7

i.x = 100
# Setting 100
```

```
i.x
# Getting ...
# 100

del i.x
# Deleting ...
i.x
# Getting ...
# 7
```

Conversiones bidireccionales

Los objetos descriptores pueden permitir que los atributos de los objetos relacionados reaccionen a los cambios automáticamente.

Supongamos que queremos modelar un oscilador con una frecuencia determinada (en hercios) y un período (en segundos). Cuando actualizamos la frecuencia, queremos que el período se actualice, y cuando actualizamos el período, queremos que la frecuencia se actualice:

```
 >>> oscillator = Oscillator(freq=100.0)  # Set frequency to 100.0 Hz
>>> oscillator.period  # Period is 1 / frequency, i.e. 0.01 seconds
0.01
>>> oscillator.period = 0.02  # Set period to 0.02 seconds
>>> oscillator.freq # The frequency is automatically adjusted
50.0
>>> oscillator.freq = 200.0  # Set the frequency to 200.0 Hz
>>> oscillator.period  # The period is automatically adjusted
0.005
```

Seleccionamos uno de los valores (frecuencia, en Hertz) como el "ancla", es decir, el que se puede establecer sin conversión, y escribimos una clase de descriptor para él:

```
class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)
```

El "otro" valor (período, en segundos) se define en términos del ancla. Escribimos una clase de descriptor que hace nuestras conversiones:

```
class Second(object):
    def __get__(self, instance, owner):
        # When reading period, convert from frequency
        return 1 / instance.freq

    def __set__(self, instance, value):
        # When setting period, update the frequency
        instance.freq = 1 / float(value)
```

Ahora podemos escribir la clase Oscilador:

```
class Oscillator(object):
    period = Second()  # Set the other value as a class attribute

    def __init__(self, freq):
        self.freq = Hertz()  # Set the anchor value as an instance attribute
        self.freq = freq  # Assign the passed value - self.period will be adjusted
```

Lea Descriptor en línea: https://riptutorial.com/es/python/topic/3405/descriptor

Capítulo 55: Despliegue

Examples

Cargando un paquete Conda

Antes de comenzar debes tener:

Anaconda instalado en su sistema Cuenta en Binstar Si no está utilizando Anaconda 1.6+, instale el cliente de línea de comandos de binstar :

```
$ conda install binstar
$ conda update binstar
```

Si no está utilizando Anaconda, Binstar también está disponible en pypi:

```
$ pip install binstar
```

Ahora podemos iniciar sesión:

```
$ binstar login
```

Prueba tu nombre de usuario con el comando whoami:

```
$ binstar whoami
```

Vamos a cargar un paquete con una función simple de "hola mundo". Para seguir, comience obteniendo mi paquete de demostración de Github:

```
$ git clone https://github.com/<NAME>/<Package>
```

Este es un pequeño directorio que se ve así:

```
package/
        setup.py
        test_package/
           __init__.py
           hello.py
           bld.bat
           build.sh
           meta.yaml
```

`Setup.py` es el archivo de compilación estándar de python y `hello.py` tiene nuestra única función hello_world ().

`bld.bat` , `build.sh` y `meta.yaml` son scripts y metadatos para el paquete `Conda` . Puede leer la página de compilación de Conda para obtener más información sobre esos tres archivos y su propósito.

Ahora creamos el paquete ejecutando:

```
$ conda build test_package/
```

Eso es todo lo que se necesita para crear un paquete Conda.

El último paso es cargar en binstar copiando y pegando la última línea de la impresión después de ejecutar el comando / paquete de conda build test. En mi sistema el comando es:

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

Como es la primera vez que creas un paquete y una versión, se te pedirá que completes algunos campos de texto que, alternativamente, se podrían hacer a través de la aplicación web.

Verá una `done` impresa para confirmar que ha cargado correctamente su paquete Conda en Binstar.

Lea Despliegue en línea: https://riptutorial.com/es/python/topic/4064/despliegue

Capítulo 56: Diccionario

Sintaxis

- mydict = {}
- mydict [k] = valor
- valor = mydict [k]
- valor = mydict.get (k)
- value = mydict.get (k, "default_value")

Parámetros

Parámetro	Detalles
llave	La clave deseada para buscar
valor	El valor a establecer o devolver.

Observaciones

Elementos útiles para recordar al crear un diccionario:

- Cada clave debe ser **única** (de lo contrario, se anulará)
- Cada tecla tiene que ser **hashable** (puede usar el `hash` la función hash que, de lo contrario `TypeError` será lanzada)
- No hay un orden particular para las llaves.

Examples

Accediendo a los valores de un diccionario.

```
dictionary = {"Hello": 1234, "World": 5678}
print(dictionary["Hello"])
```

El código anterior imprimirá `1234` .

La cadena `"Hello"` en este ejemplo se llama *clave* . Se utiliza para buscar un valor en el `dict` colocando la clave entre corchetes.

El número `1234` se ve después de los dos puntos respectivos en la definición de `dict` . Esto se llama el *valor al* que `"Hello"` *asigna* en este `dict` .

Buscar un valor como este con una clave que no existe `KeyError` una excepción `KeyError` , que `KeyError` ejecución si no se detecta. Si queremos acceder a un valor sin arriesgar un `KeyError` ,

podemos usar el método `dictionary.get` . Por defecto, si la clave no existe, el método devolverá `None` . Podemos pasarle un segundo valor para devolver en lugar de `None` en caso de una búsqueda fallida.

```
w = dictionary.get("whatever")
x = dictionary.get("whatever", "nuh-uh")
```

En este ejemplo, `w` obtendrá el valor `None` y `x` obtendrá el valor `"nuh-uh"` .

El constructor dict ()

El constructor `dict()` se puede utilizar para crear diccionarios a partir de argumentos de palabras clave, o desde una única iterable de pares clave-valor, o desde un diccionario único y argumentos de palabras clave.

```
dict(a=1, b=2, c=3)                   # {'a': 1, 'b': 2, 'c': 3}
dict([('d', 4), ('e', 5), ('f', 6)])  # {'d': 4, 'e': 5, 'f': 6}
dict([('a', 1)], b=2, c=3)            # {'a': 1, 'b': 2, 'c': 3}
dict({'a' : 1, 'b' : 2}, c=3)         # {'a': 1, 'b': 2, 'c': 3}
```

Evitar las excepciones de KeyError

Un error común cuando se usan diccionarios es acceder a una clave que no existe. Esto generalmente resulta en una excepción `KeyError`

```
mydict = {}
mydict['not there']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

Una forma de evitar errores clave es utilizar el método `dict.get` , que le permite especificar un valor predeterminado para devolver en el caso de una clave ausente.

```
value = mydict.get(key, default_value)
```

Lo que devuelve `mydict[key]` si existe, pero de lo contrario devuelve `default_value` . Tenga en cuenta que esto no agrega `key` a `mydict` . Así que si desea conservar ese par de valores clave, se debe utilizar `mydict.setdefault(key, default_value)` , lo *que* almacenar el par de valores clave.

```
mydict = {}
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}
print(mydict.setdefault("foo", "bar"))
# bar
```

```
print(mydict)
# {'foo': 'bar'}
```

Una forma alternativa de lidiar con el problema es atrapar la excepción.

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

También puede comprobar si la clave está `in` el diccionario.

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

Sin embargo, tenga en cuenta que, en entornos de subprocesos múltiples, es posible que la clave se elimine del diccionario después de la verificación, creando una condición de carrera en la que aún se puede lanzar la excepción.

Otra opción es usar una subclase de dict, collections.defaultdict, que tiene un default_factory para crear nuevas entradas en el dict cuando se le da una new_key.

Acceso a claves y valores.

Cuando se trabaja con diccionarios, a menudo es necesario acceder a todas las claves y valores del diccionario, ya sea en un bucle `for` , en una lista de comprensión, o simplemente como una lista simple.

Dado un diccionario como:

```
mydict = {
    'a': '1',
    'b': '2'
}
```

Puede obtener una lista de claves utilizando el método `keys()` :

```
print(mydict.keys())
# Python2: ['a', 'b']
# Python3: dict_keys(['b', 'a'])
```

Si, en cambio, desea una lista de valores, use el método de `values()` :

```
print(mydict.values())
# Python2: ['1', '2']
# Python3: dict_values(['2', '1'])
```

Si desea trabajar con la clave y su valor correspondiente, puede usar el método `items()` :

```
print(mydict.items())
# Python2: [('a', '1'), ('b', '2')]
# Python3: dict_items([('b', '2'), ('a', '1')])
```

NOTA: Debido a que un `dict` no está clasificado, las `keys()` , los `values()` y los `items()` no tienen orden de clasificación. Use `sort()` , `sorted()` o `OrderedDict` si le importa el orden en que regresan estos métodos.

Diferencia de Python 2/3: En Python 3, estos métodos devuelven objetos especiales iterables, no listas, y son el equivalente de los `iterkeys()` , `itervalues()` y `iteritems()` . Estos objetos se pueden usar como listas en su mayor parte, aunque hay algunas diferencias. Ver PEP 3106 para más detalles.

Introducción al Diccionario

Un diccionario es un ejemplo de un *almacén de valores clave* también conocido como *Mapeo* en Python. Le permite almacenar y recuperar elementos haciendo referencia a una clave. Como los diccionarios son referenciados por clave, tienen búsquedas muy rápidas. Como se utilizan principalmente para hacer referencia a elementos por clave, no están ordenados.

creando un dict

Los diccionarios se pueden iniciar de muchas maneras:

sintaxis literal

```
d = {}                          # empty dict
d = {'key': 'value'}            # dict with initial values
```

Python 3.x 3.5

```
# Also unpacking one or multiple dictionaries with the literal syntax is possible

# makes a shallow copy of otherdict
d = {**otherdict}
# also updates the shallow copy with the contents of the yetanotherdict.
d = {**otherdict, **yetanotherdict}
```

comprensión de dictado

```
d = {k:v for k,v in [('key', 'value',)]}
```

Ver también: Comprensiones.

clase incorporada: `dict()`

```
d = dict()                      # emtpy dict
d = dict(key='value')           # explicit keyword arguments
d = dict([('key', 'value')])  # passing in a list of key/value pairs
# make a shallow copy of another dict (only possible if keys are only strings!)
d = dict(**otherdict)
```

modificando un dict

Para agregar elementos a un diccionario, simplemente cree una nueva clave con un valor:

```
d['newkey'] = 42
```

También es posible agregar `list` y `dictionary` como valor:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

Para eliminar un elemento, elimine la clave del diccionario:

```
del d['newkey']
```

Diccionario con valores por defecto

Disponible en la biblioteca estándar como `defaultdict`

```
from collections import defaultdict

d = defaultdict(int)
d['key']                          # 0
d['key'] = 5
d['key']                          # 5

d = defaultdict(lambda: 'empty')
d['key']                          # 'empty'
d['key'] = 'full'
d['key']                          # 'full'
```

[*] Alternativamente, si debes usar la clase `dict` incorporada, `using dict.setdefault()` te permitirá crear un valor predeterminado cada vez que `using dict.setdefault()` a una clave que no existía antes:

```
>>> d = {}
{}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
{'Another_key': ['This worked!']}
```

Tenga en cuenta que si tiene muchos valores para agregar, `dict.setdefault()` creará una nueva instancia del valor inicial (en este ejemplo a `[]`) cada vez que se llame, lo que puede crear cargas

de trabajo innecesarias.

[*] *Python Cookbook, 3ª edición, por David Beazley y Brian K. Jones (O'Reilly). Derechos de autor 2013 David Beazley y Brian Jones, 978-1-449-34037-7.*

Creando un diccionario ordenado

Puede crear un diccionario ordenado que seguirá un orden determinado al iterar sobre las claves en el diccionario.

Usa `OrderedDict` del módulo de `collections` . Esto siempre devolverá los elementos del diccionario en el orden de inserción original cuando se repita.

```
from collections import OrderedDict

d = OrderedDict()
d['first'] = 1
d['second'] = 2
d['third'] = 3
d['last'] = 4

# Outputs "first 1", "second 2", "third 3", "last 4"
for key in d:
    print(key, d[key])
```

Desempaquetando diccionarios usando el operador **

Puede utilizar el operador de desempaquetado de `**` argumentos de palabras clave para entregar los pares clave-valor en un diccionario en los argumentos de una función. Un ejemplo simplificado de la documentación oficial :

```
>>>
>>> def parrot(voltage, state, action):
...     print("This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)

This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

A partir de Python 3.5, también puede utilizar esta sintaxis para fusionar un número arbitrario de objetos `dict` .

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
>>> fishdog = {**fish, **dog}
>>> fishdog

{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Como lo demuestra este ejemplo, las claves duplicadas se asignan a su último valor (por ejemplo,

"Clifford" reemplaza a "Nemo").

Fusionando diccionarios

Considere los siguientes diccionarios:

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
```

Python 3.5+

```
>>> fishdog = {**fish, **dog}
>>> fishdog
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Como lo demuestra este ejemplo, las claves duplicadas se asignan a su último valor (por ejemplo, "Clifford" reemplaza a "Nemo").

Python 3.3+

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

Con esta técnica, el valor más importante tiene prioridad para una clave dada en lugar de la última ("Clifford" se desecha a favor de "Nemo").

Python 2.x, 3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Esto utiliza el último valor, como en la técnica basada en `**` para la fusión ("Clifford" anula "Nemo").

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update` utiliza el último dict para sobrescribir el anterior.

La coma final

Al igual que las listas y las tuplas, puede incluir una coma al final en su diccionario.

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

El PEP 8 dicta que debe dejar un espacio entre la coma final y la llave de cierre.

Todas las combinaciones de valores de diccionario.

```
options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}
```

Dado un diccionario como el que se muestra arriba, donde hay una lista que representa un conjunto de valores para explorar la clave correspondiente. Supongamos que desea explorar `"x"="a"` con `"y"=10` , luego `"x"="a"` con `"y"=10` , y así sucesivamente hasta que haya explorado todas las combinaciones posibles.

Puede crear una lista que devuelva todas estas combinaciones de valores utilizando el siguiente código.

```
import itertools

options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]}

keys = options.keys()
values = (options[key] for key in keys)
combinations = [dict(zip(keys, combination)) for combination in itertools.product(*values)]
print combinations
```

Esto nos da la siguiente lista almacenada en las `combinations` variables:

```
[{'x': 'a', 'y': 10},
 {'x': 'b', 'y': 10},
 {'x': 'a', 'y': 20},
 {'x': 'b', 'y': 20},
 {'x': 'a', 'y': 30},
 {'x': 'b', 'y': 30}]
```

Iterando sobre un diccionario

Si utiliza un diccionario como iterador (por ejemplo, en una declaración `for`), atraviesa las **claves** del diccionario. Por ejemplo:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
```

```
    print(key, d[key])
# c 3
# b 2
# a 1
```

Lo mismo es cierto cuando se usa en una comprensión.

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x 3.0

El método `items()` se puede utilizar para recorrer simultáneamente tanto la **clave** como el **valor** :

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

Si bien el método de `values()` se puede usar para iterar solo sobre los valores, como se esperaría:

```
for key, value in d.values():
    print(key, value)
    # 3
    # 2
    # 1
```

Python 2.x 2.2

Aquí, las `keys()` métodos `keys()` , los `values()` y los `items()` devuelven las listas, y existen los tres métodos adicionales `iterkeys()` `itervalues()` y `iteritems()` para devolver los iteraters.

Creando un diccionario

Reglas para crear un diccionario:

- Cada clave debe ser **única** (de lo contrario, se anulará)
- Cada tecla tiene que ser **hashable** (puede usar el `hash` la función hash que, de lo contrario `TypeError` será lanzada)
- No hay un orden particular para las llaves.

```
# Creating and populating it with values
stock = {'eggs': 5, 'milk': 2}

# Or creating an empty dictionary
dictionary = {}

# And populating it after
dictionary['eggs'] = 5
dictionary['milk'] = 2

# Values can also be lists
```

```
mydict = {'a': [1, 2, 3], 'b': ['one', 'two', 'three']}

# Use list.append() method to add new elements to the values list
mydict['a'].append(4)    # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three']}
mydict['b'].append('four')  # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three', 'four']}

# We can also create a dictionary using a list of two-items tuples
iterable = [('eggs', 5), ('milk', 2)]
dictionary = dict(iterables)

# Or using keyword argument:
dictionary = dict(eggs=5, milk=2)

# Another way will be to use the dict.fromkeys:
dictionary = dict.fromkeys((milk, eggs))  # => {'milk': None, 'eggs': None}
dictionary = dict.fromkeys((milk, eggs), (2, 5))  # => {'milk': 2, 'eggs': 5}
```

Diccionarios ejemplo

Diccionarios mapean claves a valores.

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

Los valores del diccionario se pueden acceder por sus claves.

```
print "Little " + car["color"] + " " + car["model"] + "!"
# This would print out "Little Red Corvette!"
```

Los diccionarios también se pueden crear en un estilo JSON:

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

Los valores del diccionario se pueden iterar sobre:

```
for key in car:
  print key + ": " + car[key]

# wheels: 4
# color: Red
# model: Corvette
```

Lea Diccionario en línea: https://riptutorial.com/es/python/topic/396/diccionario

Capítulo 57: Diferencia entre Módulo y Paquete

Observaciones

Es posible poner un paquete de Python en un archivo ZIP, y usarlo de esa manera si agrega estas líneas al comienzo de su script:

```
import sys
sys.path.append("package.zip")
```

Examples

Módulos

Un módulo es un único archivo de Python que se puede importar. El uso de un módulo se ve así:

`module.py`

```
def hi():
    print("Hello world!")
```

`my_script.py`

```
import module
module.hi()
```

en un intérprete

```
>>> from module import hi
>>> hi()
# Hello world!
```

Paquetes

Un paquete se compone de varios archivos (o módulos) de Python e incluso puede incluir bibliotecas escritas en C o C ++. En lugar de ser un solo archivo, es una estructura de carpetas completa que podría tener este aspecto:

`package` carpetas

- `__init__.py`
- `dog.py`
- `hi.py`

`__init__.py`

```
from package.dog import woof
from package.hi import hi
```

`dog.py`

```
def woof():
    print("WOOF!!!")
```

`hi.py`

```
def hi():
    print("Hello world!")
```

Todos los paquetes de Python deben contener un archivo `__init__.py` . Cuando importa un paquete en su script (`import package`), se `__init__.py` script `__init__.py` , que le dará acceso a todas las funciones del paquete. En este caso, le permite utilizar las funciones `package.hi` y `package.woof` .

Lea Diferencia entre Módulo y Paquete en línea:
https://riptutorial.com/es/python/topic/3142/diferencia-entre-modulo-y-paquete

Capítulo 58: Distribución

Examples

py2app

Para usar el framework py2app debes instalarlo primero. Haga esto abriendo el terminal e ingresando el siguiente comando:

```
sudo easy_install -U py2app
```

También puede `pip` instalar los paquetes como:

```
pip install py2app
```

Luego crea el archivo de configuración para tu script de python:

```
py2applet --make-setup MyApplication.py
```

Edite la configuración del archivo de configuración a su gusto, esta es la predeterminada:

```
"""
This is a setup.py script generated by py2applet

Usage:
    python setup.py py2app
"""

from setuptools import setup

APP = ['test.py']
DATA_FILES = []
OPTIONS = {'argv_emulation': True}

setup(
    app=APP,
    data_files=DATA_FILES,
    options={'py2app': OPTIONS},
    setup_requires=['py2app'],
)
```

Para agregar un archivo de icono (este archivo debe tener una extensión .icns), o incluir imágenes en su aplicación como referencia, cambie las opciones como se muestra:

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

Finalmente ingrese esto en la terminal:

```
python setup.py py2app
```

El script debería ejecutarse y encontrará su aplicación terminada en la carpeta dist.

Usa las siguientes opciones para más personalización:

```
optimize (-O)          optimization level: -O1 for "python -O", -O2 for
                       "python -OO", and -O0 to disable [default: -O0]

includes (-i)          comma-separated list of modules to include

packages (-p)          comma-separated list of packages to include

extension              Bundle extension [default:.app for app, .plugin for
                       plugin]

extra-scripts          comma-separated list of additional scripts to include
                       in an application or plugin.
```

cx_Freeze

Instala cx_Freeze desde aquí

Descomprima la carpeta y ejecute estos comandos desde ese directorio:

```
python setup.py build
sudo python setup.py install
```

Cree un nuevo directorio para su script de python y cree un archivo **"setup.py"** en el mismo directorio con el siguiente contenido:

```
application_title = "My Application" # Use your own application name
main_python_file = "my_script.py" # Your python script

import sys

from cx_Freeze import setup, Executable

base = None
if sys.platform == "win32":
    base = "Win32GUI"

includes = ["atexit","re"]

setup(
        name = application_title,
        version = "0.1",
        description = "Your Description",
        options = {"build_exe" : {"includes" : includes }},
        executables = [Executable(main_python_file, base = base)])
```

Ahora ejecuta tu setup.py desde la terminal:

```
python setup.py bdist_mac
```

NOTA: En El Capitán, esto deberá ejecutarse como root con el modo SIP deshabilitado.

Lea Distribución en línea: https://riptutorial.com/es/python/topic/2026/distribucion

Capítulo 59: Django

Introducción

Django es un marco web de Python de alto nivel que fomenta el desarrollo rápido y el diseño limpio y pragmático. Creado por desarrolladores experimentados, se encarga de gran parte de la molestia del desarrollo web, por lo que puede centrarse en escribir su aplicación sin necesidad de reinventar la rueda. Es gratis y de código abierto.

Examples

Hola mundo con django

Haz un ejemplo simple de `Hello World` usando tu django.

Asegurémonos de que tienes django instalado en tu PC primero.

abre una terminal y escribe: python -c "import django"
-> si no aparece ningún error, significa que django ya está instalado.

Ahora vamos a crear un proyecto en django. Para eso escribe abajo el comando en la terminal:
django-admin startproject HelloWorld

El comando anterior creará un directorio llamado HelloWorld.
La estructura del directorio será como:
Hola Mundo
|
| | - **init** .py
| | --settings.py
| | --urls.py
| | --wsgi.py
| --manejar.py

Escritura de vistas (Referencia de la documentación de django)

Una función de vista, o vista para abreviar, es simplemente una función de Python que toma una solicitud web y devuelve una respuesta web. Esta respuesta puede ser el contenido HTML de una página web o cualquier cosa. La documentación dice que podemos escribir la función de vistas en cualquier lugar, pero es mejor escribir en views.py ubicado en nuestro directorio de proyectos.

Aquí hay una vista que devuelve un mensaje de hello world. (Views.py)

```
from django.http import HttpResponse

define helloWorld(request):
    return HttpResponse("Hello World!! Django Welcomes You.")
```

entendamos el código, paso a paso.

- Primero, importamos la clase HttpResponse desde el módulo django.http.
- A continuación, definimos una función llamada helloworld. Esta es la función de vista. Cada función de vista toma un objeto HttpRequest como su primer parámetro, que normalmente se denomina solicitud.

  Tenga en cuenta que el nombre de la función de vista no importa; no tiene que ser nombrado de cierta manera para que Django lo reconozca. Lo llamamos helloworld aquí, para que quede claro lo que hace.
- La vista devuelve un objeto HttpResponse que contiene la respuesta generada. Cada función de vista es responsable de devolver un objeto HttpResponse.

Para más información sobre las vistas de Django, haga clic aquí.

Mapeo de URLs a vistas
Para mostrar esta vista en una URL particular, deberá crear una URLconf;

Antes de eso vamos a entender cómo django procesa las solicitudes.

- Django determina el módulo raíz URLconf a usar.
- Django carga ese módulo de Python y busca las variables urlpatterns. Esta debería ser una lista de Python de instancias de django.conf.urls.url ().
- Django recorre cada patrón de URL, en orden, y se detiene en el primero que coincide con la URL solicitada.
- Una vez que una de las expresiones regulares coincide, Django importa y llama a la vista dada, que es una función simple de Python.

Así es como se ve nuestro URLconf:

```
from django.conf.urls import url
from . import views #import the views.py from current directory

urlpatterns = [
   url(r'^helloworld/$', views.helloWorld),
]
```

Para más información sobre las URL de Django, haga clic aquí.

Ahora cambie el directorio a HelloWorld y escriba el comando siguiente en la terminal.
python manage.py runserver

de forma predeterminada, el servidor se ejecutará en 127.0.0.1:8000

Abra su navegador y escriba 127.0.0.1:8000/helloworld/. La página te mostrará "¡Hola mundo! Django te da la bienvenida".

Lea Django en línea: https://riptutorial.com/es/python/topic/8994/django

Capítulo 60: Ejecución de código dinámico con `exec` y` eval`

Sintaxis

- eval (expresión [, globals = None [, locals = None]])
- exec (objeto)
- exec (objeto, globals)
- exec (objeto, globales, locales)

Parámetros

Argumento	Detalles
`expression`	El código de expresión como una cadena o un objeto de `code`
`object`	El código de la declaración como una cadena, o un objeto de `code`
`globals`	El diccionario a utilizar para variables globales. Si no se especifican los locales, esto también se usa para los locales. Si se omite, se utilizan los `globals()` del ámbito de llamada.
`locals`	Un objeto de *mapeo* que se utiliza para las variables locales. Si se omite, se usa el que se pasa para los `globals` . Si se omiten ambos, entonces se utilizan los `globals()` y `locals()` del ámbito de llamada para los `globals` y `locals` respectivamente.

Observaciones

En `exec` , si los `globals` son `locals` (es decir, se refieren al mismo objeto), el código se ejecuta como si estuviera en el nivel del módulo. Si los elementos `globals` y `locals` son objetos distintos, el código se ejecuta como si estuviera en un *cuerpo de clase* .

Si las `globals` de objeto se pasa en, pero no especifica `__builtins__` clave, entonces Python funciones integradas y los nombres se añaden automáticamente al ámbito global. Para suprimir la disponibilidad de funciones como la `print` o la `isinstance` en el ámbito ejecutado, permita que los `globals` tengan la clave `__builtins__` asignada al valor `None` . Sin embargo, esto no es una característica de seguridad.

La sintaxis específica de Python 2 no debe usarse; La sintaxis de Python 3 funcionará en Python 2. Por lo tanto, los siguientes formularios están en desuso: <s>

- `exec object`
- `exec object in globals`

- `exec object in globals, locals`

Examples

Evaluando declaraciones con exec

```
>>> code = """for i in range(5):\n    print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Evaluando una expresión con eval

```
>>> expression = '5 + 3 * a'
>>> a = 5
>>> result = eval(expression)
>>> result
20
```

Precompilando una expresión para evaluarla varias veces.

`compile` función incorporada de `compile` se puede utilizar para precompilar una expresión en un objeto de código; este objeto de código se puede pasar a eval. Esto acelerará las ejecuciones repetidas del código evaluado. El tercer parámetro para `compile` debe ser la cadena `'eval'` .

```
>>> code = compile('a * b + c', '<string>', 'eval')
>>> code
<code object <module> at 0x7f0e51a58830, file "<string>", line 1>
>>> a, b, c = 1, 2, 3
>>> eval(code)
5
```

Evaluar una expresión con eval utilizando globales personalizados

```
>>> variables = {'a': 6, 'b': 7}
>>> eval('a * b', globals=variables)
42
```

Como un plus, con esto el código no puede referirse accidentalmente a los nombres definidos fuera:

```
>>> eval('variables')
{'a': 6, 'b': 7}
>>> eval('variables', globals=variables)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
```

```
NameError: name 'variables' is not defined
```

El uso de `defaultdict` permite, por ejemplo, tener variables indefinidas configuradas en cero:

```
>>> from collections import defaultdict
>>> variables = defaultdict(int, {'a': 42})
>>> eval('a * c', globals=variables)  # note that 'c' is not explicitly defined
0
```

Evaluar una cadena que contiene un literal de Python con ast.literal_eval

Si tiene una cadena que contiene literales de Python, como cadenas, flotadores, etc., puede usar `ast.literal_eval` para evaluar su valor en lugar de `eval` . Esto tiene la característica adicional de permitir solo cierta sintaxis.

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

Sin embargo, esto no es seguro para la ejecución del código proporcionado por un usuario no confiable, y es trivial bloquear un intérprete con una entrada cuidadosamente diseñada

```
>>> import ast
>>> ast.literal_eval('()' * 1000000)
[5]    21358 segmentation fault (core dumped)  python3
```

Aquí, la entrada es una cadena de `()` repetida un millón de veces, lo que provoca un bloqueo en el analizador CPython. Los desarrolladores de CPython no consideran los errores en el analizador como problemas de seguridad.

Código de ejecución proporcionado por un usuario no confiable que utiliza exec, eval o ast.literal_eval

No es posible usar `eval` o `exec` para ejecutar código de un usuario no confiable de forma segura. Incluso `ast.literal_eval` es propenso a bloqueos en el analizador. A veces es posible protegerse contra la ejecución de código malicioso, pero no excluye la posibilidad de bloqueos directos en el analizador o el tokenizador.

Para evaluar el código por un usuario que no es de confianza, debe recurrir a algún módulo de terceros, o tal vez escribir su propio analizador y su propia máquina virtual en Python.

Lea Ejecución de código dinámico con `exec` y` eval` en línea:
https://riptutorial.com/es/python/topic/2251/ejecucion-de-codigo-dinamico-con--exec--y--eval-

Capítulo 61: El dis módulo

Examples

Constantes en el módulo dis

```
EXTENDED_ARG = 145 # All opcodes greater than this have 2 operands
HAVE_ARGUMENT = 90 # All opcodes greater than this have at least 1 operands

cmp_op = ('<', '<=', '==', '!=', '>', '>=', 'in', 'not in', 'is', 'is ...
       # A list of comparator id's. The indecies are used as operands in some opcodes

# All opcodes in these lists have the respective types as there operands
hascompare = [107]
hasconst = [100]
hasfree = [135, 136, 137]
hasjabs = [111, 112, 113, 114, 115, 119]
hasjrel = [93, 110, 120, 121, 122, 143]
haslocal = [124, 125, 126]
hasname = [90, 91, 95, 96, 97, 98, 101, 106, 108, 109, 116]

# A map of opcodes to ids
opmap = {'BINARY_ADD': 23, 'BINARY_AND': 64, 'BINARY_DIVIDE': 21, 'BIN...
# A map of ids to opcodes
opname = ['STOP_CODE', 'POP_TOP', 'ROT_TWO', 'ROT_THREE', 'DUP_TOP', '...
```

¿Qué es el código de bytes de Python?

Python es un intérprete híbrido. Al ejecutar un programa, primero lo ensambla en un *código de bytes* que luego puede ejecutarse en el intérprete de Python (también denominado *máquina virtual de Python*). El módulo `dis` en la biblioteca estándar se puede usar para hacer que el bytecode de Python sea legible al desensamblar clases, métodos, funciones y objetos de código.

```
>>> def hello():
...     print "Hello, World"
...
>>> dis.dis(hello)
  2           0 LOAD_CONST               1 ('Hello, World')
              3 PRINT_ITEM
              4 PRINT_NEWLINE
              5 LOAD_CONST               0 (None)
              8 RETURN_VALUE
```

El intérprete de Python se basa en la pila y utiliza un sistema de último en entrar, último en salir.

Cada código de operación (código de operación) en el lenguaje ensamblador de Python (el bytecode) toma un número fijo de elementos de la pila y devuelve un número fijo de elementos a la pila. Si no hay suficientes elementos en la pila para un código de operación, el intérprete de Python se bloqueará, posiblemente sin un mensaje de error.

Desmontaje de módulos.

Para desensamblar un módulo de Python, primero se debe convertir en un archivo `.pyc` (compilado por Python). Para hacer esto, corre

```
python -m compileall <file>.py
```

Luego, en un intérprete, ejecute

```
import dis
import marshal
with open("<file>.pyc", "rb") as code_f:
    code_f.read(8) # Magic number and modification time
    code = marshal.load(code_f) # Returns a code object which can be disassembled
    dis.dis(code) # Output the disassembly
```

Esto compilará un módulo de Python y dará salida a las instrucciones del código de bytes con `dis` . El módulo nunca se importa, por lo que es seguro utilizarlo con código no confiable.

Lea El dis módulo en línea: https://riptutorial.com/es/python/topic/1763/el-dis-modulo

Capítulo 62: El intérprete (consola de línea de comandos)

Examples

Obtención de ayuda general

Si se llama a la función de `help` en la consola sin ningún argumento, Python presenta una consola de ayuda interactiva, donde puede obtener información sobre módulos, símbolos, palabras clave y más de Python.

```
>>> help()

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.4/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".
```

Refiriéndose a la última expresión.

Para obtener el valor del último resultado de su última expresión en la consola, use un guión bajo `_` .

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

Este valor de subrayado mágico solo se actualiza cuando se usa una expresión de python que da como resultado un valor. Definir funciones o para bucles no cambia el valor. Si la expresión genera una excepción, no habrá cambios en `_` .

```
>>> "Hello, {0}".format("World")
'Hello, World'
>>> _
'Hello, World'
>>> def wontchangethings():
...     pass
```

```
>>> _
'Hello, World'
>>> 27 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> _
'Hello, World'
```

Recuerde, esta variable mágica solo está disponible en el intérprete interactivo de Python. Ejecutar scripts no hará esto.

Abriendo la consola de Python

La consola para la versión principal de Python por lo general se puede abrir escribiendo `py` en la consola de Windows o `python` en otras plataformas.

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Si tiene varias versiones, entonces, de forma predeterminada, sus ejecutables se asignarán a `python2` o `python3` respectivamente.

Por supuesto, esto depende de que los ejecutables de Python estén en tu RUTA.

La variable PYTHONSTARTUP

Puede establecer una variable de entorno llamada PYTHONSTARTUP para la consola de Python. Cada vez que ingrese a la consola de Python, este archivo se ejecutará, lo que le permitirá agregar funcionalidad adicional a la consola, como la importación automática de módulos de uso común.

Si la variable PYTHONSTARTUP se configuró en la ubicación de un archivo que contiene esto:

```
print("Welcome!")
```

Luego, abrir la consola de Python resultaría en esta salida adicional:

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Welcome!
>>>
```

Argumentos de línea de comando

Python tiene una variedad de interruptores de línea de comando que se pueden pasar a `py` . Se pueden encontrar ejecutando `py --help` , que proporciona esta salida en Python 3.4:

```
Python Launcher

usage: py [ launcher-arguments ] [ python-arguments ] script [ script-arguments ]

Launcher arguments:

-2     : Launch the latest Python 2.x version
-3     : Launch the latest Python 3.x version
-X.Y   : Launch the specified Python version
-X.Y-32: Launch the specified 32bit Python version

The following help text is from Python:

usage: G:\Python34\python.exe [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I     : isolate Python from the user's environment (implies -E and -s)
-m mod : run library module as a script (terminates option list)
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO    : remove doc-strings in addition to the -O optimizations
-q     : don't print version and copyright messages on interactive startup
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-u     : unbuffered binary stdout and stderr, stdin always buffered;
         also PYTHONUNBUFFERED=x
         see man page for details on internal buffering relating to '-u'
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
-W arg : warning control; arg is action:message:category:module:lineno
         also PYTHONWARNINGS=arg
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt : set implementation-specific option
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ';'-separated list of directories prefixed to the
               default module search path.  The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>;<exec_prefix>).
               The default module search path uses <prefix>\lib.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
PYTHONHASHSEED: if this variable is set to 'random', a random value is used
   to seed the hashes of str, bytes and datetime objects.  It can also be
   set to an integer in the range [0,4294967295] to get hash values with a
   predictable seed.
```

Obteniendo ayuda sobre un objeto

La consola de Python agrega una nueva función, `help` , que se puede usar para obtener información sobre una función u objeto.

Para una función, la `help` imprime su firma (argumentos) y su cadena de documentación, si la función tiene una.

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Para un objeto, la `help` enumera la cadena de documentos del objeto y las diferentes funciones miembro que tiene el objeto.

```
>>> x = 2
>>> help(x)
Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value...
```

Lea El intérprete (consola de línea de comandos) en línea: https://riptutorial.com/es/python/topic/2473/el-interprete--consola-de-linea-de-comandos-

Capítulo 63: El módulo base64

Introducción

La codificación de Base 64 representa un esquema común para la codificación de binarios en formato de cadena ASCII utilizando radix 64. El módulo base64 es parte de la biblioteca estándar, lo que significa que se instala junto con Python. La comprensión de los bytes y las cadenas es fundamental para este tema y se puede revisar aquí . Este tema explica cómo usar las distintas funciones y bases numéricas del módulo base64.

Sintaxis

- base64.b64encode (s, altchars = Ninguno)
- base64.b64decode (s, altchars = None, validate = False)
- base64.standard_b64encode (s)
- base64.standard_b64decode (s)
- base64.urlsafe_b64encode (s)
- base64.urlsafe_b64decode (s)
- base64.b32encode (s)
- base64.b32decode (s)
- base64.b16encode (s)
- base64.b16decode (s)
- base64.a85encode (b, foldspaces = False, wrapcol = 0, pad = False, adobe = False)
- base64.a85decode (b, foldpaces = False, adobe = False, ignorechars = b '\ t \ n \ r \ v')
- base64.b85encode (b, pad = False)
- base64.b85decode (b)

Parámetros

Parámetro	Descripción
`base64.b64encode(s, altchars=None)`	
s	Un objeto parecido a bytes
altares	Un objeto similar a bytes de longitud 2+ de caracteres para reemplazar los caracteres '+' y '=' al crear el alfabeto Base64. Los caracteres adicionales son ignorados.
`base64.b64decode(s, altchars=None, validate=False)`	
s	Un objeto parecido a bytes
altares	Un objeto similar a bytes de longitud 2+ de

Parámetro	Descripción
	caracteres para reemplazar los caracteres '+' y '=' al crear el alfabeto Base64. Los caracteres adicionales son ignorados.
validar	Si valide es True, los caracteres que no están en el alfabeto Base64 normal o en el alfabeto alternativo **no se** descartan antes de la verificación de relleno
`base64.standard_b64encode(s)`	
s	Un objeto parecido a bytes
`base64.standard_b64decode(s)`	
s	Un objeto parecido a bytes
`base64.urlsafe_b64encode(s)`	
s	Un objeto parecido a bytes
`base64.urlsafe_b64decode(s)`	
s	Un objeto parecido a bytes
`b32encode(s)`	
s	Un objeto parecido a bytes
`b32decode(s)`	
s	Un objeto parecido a bytes
`base64.b16encode(s)`	
s	Un objeto parecido a bytes
`base64.b16decode(s)`	
s	Un objeto parecido a bytes
`base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)`	
segundo	Un objeto parecido a bytes
espacios de plegado	Si foldspaces es True, se utilizará el carácter 'y' en lugar de 4 espacios consecutivos.
envoltura	Los caracteres numéricos antes de una nueva línea (0 implica que no hay nuevas líneas)
almohadilla	Si el pad es True, los bytes se rellenan a un

Parámetro	Descripción
	múltiplo de 4 antes de codificar
adobe	Si Adobe es verdadero, la secuencia codificada se debe enmarcar con '<~' y '' ~> 'como se usa con los productos de Adobe.
`base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')`	
segundo	Un objeto parecido a bytes
espacios de plegado	Si foldspaces es True, se utilizará el carácter 'y' en lugar de 4 espacios consecutivos.
adobe	Si Adobe es verdadero, la secuencia codificada se debe enmarcar con '<~' y '' ~> 'como se usa con los productos de Adobe.
ignorantes	Un objeto de caracteres similar a bytes que se ignorará en el proceso de codificación.
`base64.b85encode(b, pad=False)`	
segundo	Un objeto parecido a bytes
almohadilla	Si el pad es True, los bytes se rellenan a un múltiplo de 4 antes de codificar
`base64.b85decode(b)`	
segundo	Un objeto parecido a bytes

Observaciones

Hasta que salió Python 3.4, las funciones de codificación y decodificación de base64 solo funcionaban con `bytes` o tipos de `bytearray` . Ahora estas funciones aceptan cualquier objeto similar a bytes .

Examples

Codificación y decodificación Base64

Para incluir el módulo base64 en su script, primero debe importarlo:

```
import base64
```

Las funciones de codificación y decodificación de base64 requieren un objeto similar a bytes .

Para convertir nuestra cadena en bytes, debemos codificarla utilizando la función de codificación incorporada de Python. Más comúnmente, se usa la codificación `UTF-8` , sin embargo, se puede encontrar una lista completa de estas codificaciones estándar (incluidos los idiomas con diferentes caracteres) aquí en la Documentación oficial de Python. A continuación se muestra un ejemplo de codificación de una cadena en bytes:

```
s = "Hello World!"
b = s.encode("UTF-8")
```

La salida de la última línea sería:

```
b'Hello World!'
```

El prefijo `b` se utiliza para indicar que el valor es un objeto de bytes.

Para codificar en Base64 estos bytes, usamos la función `base64.b64encode()` :

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
print(e)
```

Ese código daría como resultado lo siguiente:

```
b'SGVsbG8gV29ybGQh'
```

que todavía está en el objeto bytes. Para obtener una cadena de estos bytes, podemos usar el método `decode()` Python con la `UTF-8` :

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
s1 = e.decode("UTF-8")
print(s1)
```

La salida sería entonces:

```
SGVsbG8gV29ybGQh
```

Si quisiéramos codificar la cadena y luego decodificar, podríamos usar el método `base64.b64decode()` :

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base64 Encode the bytes
e = base64.b64encode(b)
# Decoding the Base64 bytes to string
s1 = e.decode("UTF-8")
# Printing Base64 encoded string
```

```
print("Base64 Encoded:", s1)
# Encoding the Base64 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base64 bytes
d = base64.b64decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Como es de esperar, la salida sería la cadena original:

```
Base64 Encoded: SGVsbG8gV29ybGQh
Hello World!
```

Codificación y decodificación Base32

El módulo base64 también incluye funciones de codificación y decodificación para Base32. Estas funciones son muy similares a las funciones de Base64:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base32 Encode the bytes
e = base64.b32encode(b)
# Decoding the Base32 bytes to string
s1 = e.decode("UTF-8")
# Printing Base32 encoded string
print("Base32 Encoded:", s1)
# Encoding the Base32 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base32 bytes
d = base64.b32decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Esto produciría el siguiente resultado:

```
Base32 Encoded: JBSWY3DPEBLW64TMMQQQ====
Hello World!
```

Codificación y decodificación Base16

El módulo base64 también incluye funciones de codificación y decodificación para Base16. La base 16 es comúnmente conocida como **hexadecimal** . Estas funciones son muy similares a las funciones Base64 y Base32:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
```

```
b = s.encode("UTF-8")
# Base16 Encode the bytes
e = base64.b16encode(b)
# Decoding the Base16 bytes to string
s1 = e.decode("UTF-8")
# Printing Base16 encoded string
print("Base16 Encoded:", s1)
# Encoding the Base16 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base16 bytes
d = base64.b16decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Esto produciría el siguiente resultado:

```
Base16 Encoded: 48656C6C6F20576F726C6421
Hello World!
```

Codificación y decodificación ASCII85

Adobe creó su propia codificación llamada **ASCII85** que es similar a Base85, pero tiene sus diferencias. Esta codificación se utiliza con frecuencia en los archivos PDF de Adobe. Estas funciones fueron lanzadas en la versión 3.4 de Python. De lo contrario, las funciones `base64.a85encode()` y `base64.a85encode()` son similares a las anteriores:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# ASCII85 Encode the bytes
e = base64.a85encode(b)
# Decoding the ASCII85 bytes to string
s1 = e.decode("UTF-8")
# Printing ASCII85 encoded string
print("ASCII85 Encoded:", s1)
# Encoding the ASCII85 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the ASCII85 bytes
d = base64.a85decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Esto da como resultado lo siguiente:

```
ASCII85 Encoded: 87cURD]i,"Ebo80
Hello World!
```

Codificación y decodificación Base85

Al igual que las funciones Base64, Base32 y Base16, las funciones de codificación y

decodificación `base64.b85encode()` son `base64.b85encode()` y `base64.b85decode()` :

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base85 Encode the bytes
e = base64.b85encode(b)
# Decoding the Base85 bytes to string
s1 = e.decode("UTF-8")
# Printing Base85 encoded string
print("Base85 Encoded:", s1)
# Encoding the Base85 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base85 bytes
d = base64.b85decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

que produce lo siguiente:

```
Base85 Encoded: NM&qnZy;B1a%^NF
Hello World!
```

Lea El módulo base64 en línea: https://riptutorial.com/es/python/topic/8678/el-modulo-base64

Capítulo 64: El módulo de configuración regional

Observaciones

Python 2 Docs: [https://docs.python.org/2/library/locale.html#locale.currency◆ [ ]]

Examples

Formato de moneda Dólares estadounidenses utilizando el módulo de configuración regional

```
import locale

locale.setlocale(locale.LC_ALL, '')
Out[2]: 'English_United States.1252'

locale.currency(762559748.49)
Out[3]: '$762559748.49'

locale.currency(762559748.49, grouping=True)
Out[4]: '$762,559,748.49'
```

Lea El módulo de configuración regional en línea: https://riptutorial.com/es/python/topic/1783/el-modulo-de-configuracion-regional

Capítulo 65: El módulo os

Introducción

Este módulo proporciona una forma portátil de utilizar la funcionalidad dependiente del sistema operativo.

Sintaxis

- importación OS

Parámetros

Parámetro	Detalles
Camino	Una ruta a un archivo. El separador de ruta puede ser determinado por `os.path.sep` .
Modo	El permiso deseado, en octal (por ejemplo, `0700`)

Examples

Crear un directorio

```
os.mkdir('newdir')
```

Si necesita especificar permisos, puede usar el argumento de `mode` opcional:

```
os.mkdir('newdir', mode=0700)
```

Obtener directorio actual

Utilice la función `os.getcwd()` :

```
print(os.getcwd())
```

Determinar el nombre del sistema operativo.

El módulo `os` proporciona una interfaz para determinar en qué tipo de sistema operativo se está ejecutando actualmente el código.

```
os.name
```

Esto puede devolver uno de los siguientes en Python 3:

- `posix`
- `nt`
- `ce`
- `java`

Se puede obtener información más detallada de `sys.platform`

Eliminar un directorio

Eliminar el directorio en la `path` :

```
os.rmdir(path)
```

No debe usar `os.remove()` para eliminar un directorio. Esa función es para *archivos* y su uso en directorios resultará en un `OSError`

Seguir un enlace simbólico (POSIX)

A veces es necesario determinar el objetivo de un enlace simbólico. `os.readlink` hará esto:

```
print(os.readlink(path_to_symlink))
```

Cambiar permisos en un archivo

```
os.chmod(path, mode)
```

donde `mode` es el permiso deseado, en octal.

makedirs - creación de directorio recursivo

Dado un directorio local con los siguientes contenidos:

```
└── dir1
    ├── subdir1
    └── subdir2
```

Queremos crear el mismo subdir1, subdir2 bajo un nuevo directorio dir2, que aún no existe.

```
import os

os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

Ejecutando estos resultados en

```
├── dir1
│   ├── subdir1
```

```
│   └── subdir2
└── dir2
    ├── subdir1
    └── subdir2
```

dir2 solo se crea la primera vez que se necesita, para la creación de subdir1.

Si hubiéramos usado **os.mkdir** en **su** lugar, habríamos tenido una excepción porque dir2 no habría existido todavía.

```
    os.mkdir("./dir2/subdir1")
OSError: [Errno 2] No such file or directory: './dir2/subdir1'
```

os.makedirs no le gustará si el directorio de destino ya existe. Si lo volvemos a ejecutar de nuevo:

```
OSError: [Errno 17] File exists: './dir2/subdir1'
```

Sin embargo, esto podría solucionarse fácilmente detectando la excepción y comprobando que el directorio se haya creado.

```
try:
    os.makedirs("./dir2/subdir1")
except OSError:
    if not os.path.isdir("./dir2/subdir1"):
        raise

try:
    os.makedirs("./dir2/subdir2")
except OSError:
    if not os.path.isdir("./dir2/subdir2"):
        raise
```

Lea El módulo os en línea: https://riptutorial.com/es/python/topic/4127/el-modulo-os

Capítulo 66: Empezando con GZip

Introducción

Este módulo proporciona una interfaz simple para comprimir y descomprimir archivos al igual que los programas GNU gzip y gunzip.

La compresión de datos es proporcionada por el módulo zlib.

El módulo gzip proporciona la clase GzipFile que se modela después del objeto File de Python. La clase GzipFile lee y escribe archivos en formato gzip, comprimiendo o descomprimiendo automáticamente los datos para que se parezca a un objeto de archivo normal.

Examples

Lee y escribe archivos zip de GNU

```
import gzip
import os

outfilename = 'example.txt.gz'
output = gzip.open(outfilename, 'wb')
try:
    output.write('Contents of the example file go here.\n')
finally:
    output.close()

print outfilename, 'contains', os.stat(outfilename).st_size, 'bytes of compressed data'
os.system('file -b --mime %s' % outfilename)
```

Guárdelo como 1gzip_write.py1. Ejecútelo a través del terminal.

```
$ python gzip_write.py

application/x-gzip; charset=binary
example.txt.gz contains 68 bytes of compressed data
```

Lea Empezando con GZip en línea: https://riptutorial.com/es/python/topic/8993/empezando-con-gzip

Capítulo 67: Enchufes

Introducción

Muchos lenguajes de programación usan sockets para comunicarse a través de procesos o entre dispositivos. Este tema explica el uso correcto del módulo de sockets en Python para facilitar el envío y la recepción de datos a través de protocolos de red comunes.

Parámetros

Parámetro	Descripción
socket.AF_UNIX	Unix Socket
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

Examples

Envío de datos a través de UDP

UDP es un protocolo sin conexión. Los mensajes a otros procesos o computadoras se envían sin establecer ningún tipo de conexión. No hay confirmación automática si su mensaje ha sido recibido. UDP se utiliza generalmente en aplicaciones sensibles a la latencia o en aplicaciones que envían transmisiones de toda la red.

El siguiente código envía un mensaje a un proceso que escucha en el puerto de host local 6667 usando UDP

Tenga en cuenta que no es necesario "cerrar" el socket después del envío, ya que UDP no tiene conexión .

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
msg = ("Hello you there!").encode('utf-8')  # socket.sendto() takes bytes as input, hence we
must encode the string first.
s.sendto(msg, ('localhost', 6667))
```

Recepción de datos a través de UDP

UDP es un protocolo sin conexión. Esto significa que los pares que envían mensajes no requieren establecer una conexión antes de enviar mensajes. `socket.recvfrom` devuelve una tupla (`msg` [el mensaje que recibió el socket], `addr` [la dirección del remitente])

Un servidor UDP que utiliza únicamente el módulo de `socket` :

```
from socket import socket, AF_INET, SOCK_DGRAM
sock = socket(AF_INET, SOCK_DGRAM)
sock.bind(('localhost', 6667))

while True:
    msg, addr = sock.recvfrom(8192)  # This is the amount of bytes to read at maximum
    print("Got message from %s: %s" % (addr, msg))
```

A continuación se muestra una implementación alternativa utilizando `socketserver.UDPServer` :

```
from socketserver import BaseRequestHandler, UDPServer

class MyHandler(BaseRequestHandler):
    def handle(self):
        print("Got connection from: %s" % self.client_address)
        msg, sock = self.request
        print("It said: %s" % msg)
        sock.sendto("Got your message!".encode(), self.client_address) # Send reply

serv = UDPServer(('localhost', 6667), MyHandler)
serv.serve_forever()
```

Por defecto, los `sockets` bloquean. Esto significa que la ejecución del script esperará hasta que el socket reciba datos.

Envío de datos a través de TCP

El envío de datos a través de Internet es posible mediante múltiples módulos. El módulo de sockets proporciona acceso de bajo nivel a las operaciones subyacentes del sistema operativo responsables de enviar o recibir datos de otras computadoras o procesos.

El siguiente código envía la cadena de bytes `b'Hello'` a un servidor TCP que escucha en el puerto 6667 en el host local y cierra la conexión cuando termina:

```
from socket import socket, AF_INET, SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 6667))  # The address of the TCP server listening
s.send(b'Hello')
s.close()
```

La salida del zócalo está bloqueando de forma predeterminada, lo que significa que el programa esperará en la conexión y enviará las llamadas hasta que la acción se complete. Para la conexión eso significa que el servidor realmente acepta la conexión. Para enviar, solo significa que el sistema operativo tiene suficiente espacio de almacenamiento para poner en cola los datos que se enviarán más tarde.

Los enchufes siempre deben estar cerrados después de su uso.

Servidor de socket TCP multihilo

Cuando se ejecuta sin argumentos, este programa inicia un servidor de socket TCP que escucha las conexiones a `127.0.0.1` en el puerto `5000` . El servidor maneja cada conexión en un hilo separado.

Cuando se ejecuta con el argumento `-c` , este programa se conecta al servidor, lee la lista de clientes y la imprime. La lista de clientes se transfiere como una cadena JSON. El nombre del cliente puede especificarse pasando el argumento `-n` . Al pasar nombres diferentes, se puede observar el efecto en la lista de clientes.

client_list.py

```
import argparse
import json
import socket
import threading

def handle_client(client_list, conn, address):
    name = conn.recv(1024)
    entry = dict(zip(['name', 'address', 'port'], [name, address[0], address[1]]))
    client_list[name] = entry
    conn.sendall(json.dumps(client_list))
    conn.shutdown(socket.SHUT_RDWR)
    conn.close()

def server(client_list):
    print "Starting server..."
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind(('127.0.0.1', 5000))
    s.listen(5)
    while True:
        (conn, address) = s.accept()
        t = threading.Thread(target=handle_client, args=(client_list, conn, address))
        t.daemon = True
        t.start()

def client(name):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('127.0.0.1', 5000))
    s.send(name)
    data = s.recv(1024)
    result = json.loads(data)
    print json.dumps(result, indent=4)

def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', dest='client', action='store_true')
    parser.add_argument('-n', dest='name', type=str, default='name')
    result = parser.parse_args()
    return result

def main():
    client_list = dict()
```

```
    args = parse_arguments()
    if args.client:
        client(args.name)
    else:
        try:
            server(client_list)
        except KeyboardInterrupt:
            print "Keyboard interrupt"

if __name__ == '__main__':
    main()
```

Salida del servidor

```
$ python client_list.py
Starting server...
```

Salida de cliente

```
$ python client_list.py -c -n name1
{
    "name1": {
        "address": "127.0.0.1",
        "port": 62210,
        "name": "name1"
    }
}
```

Los buffers de recepción están limitados a 1024 bytes. Si la representación de la cadena JSON de la lista de clientes supera este tamaño, se truncará. Esto hará que se genere la siguiente excepción:

```
ValueError: Unterminated string starting at: line 1 column 1023 (char 1022)
```

Raw Sockets en Linux

Primero desactivas la suma de comprobación automática de tu tarjeta de red:

```
sudo ethtool -K eth1 tx off
```

Luego envíe su paquete, utilizando un socket SOCK_RAW:

```
#!/usr/bin/env python
from socket import socket, AF_PACKET, SOCK_RAW
s = socket(AF_PACKET, SOCK_RAW)
s.bind(("eth1", 0))

# We're putting together an ethernet frame here,
# but you could have anything you want instead
# Have a look at the 'struct' module for more
# flexible packing/unpacking of binary data
# and 'binascii' for 32 bit CRC
src_addr = "\x01\x02\x03\x04\x05\x06"
```

```
dst_addr = "\x01\x02\x03\x04\x05\x06"
payload = ("["*30)+"PAYLOAD"+("]"*30)
checksum = "\x1a\x2b\x3c\x4d"
ethertype = "\x08\x01"

s.send(dst_addr+src_addr+ethertype+payload+checksum)
```

Lea Enchufes en línea: https://riptutorial.com/es/python/topic/1530/enchufes

Capítulo 68: entorno virtual con virtualenvwrapper

Introducción

Supongamos que necesita trabajar en tres proyectos diferentes. El proyecto A, el proyecto B y el proyecto C. el proyecto A y el proyecto B necesitan Python 3 y algunas bibliotecas requeridas. Pero para el proyecto C necesitas Python 2.7 y bibliotecas dependientes.

Así que la mejor práctica para esto es separar esos entornos de proyecto. Para crear un entorno virtual puedes usar la siguiente técnica:

Virtualenv, Virtualenvwrapper y Conda

Aunque tenemos varias opciones para el entorno virtual, se recomienda virtualenvwrapper.

Examples

Crear entorno virtual con virtualenvwrapper

Supongamos que necesita trabajar en tres proyectos diferentes. El proyecto A, el proyecto B y el proyecto C. el proyecto A y el proyecto B necesitan Python 3 y algunas bibliotecas requeridas. Pero para el proyecto C necesitas Python 2.7 y bibliotecas dependientes.

Así que la mejor práctica para esto es separar esos entornos de proyecto. Para crear un entorno virtual puedes usar la siguiente técnica:

Virtualenv, Virtualenvwrapper y Conda

Aunque tenemos varias opciones para el entorno virtual, se recomienda virtualenvwrapper.

Aunque tenemos varias opciones para el entorno virtual, siempre prefiero virtualenvwrapper porque tiene más facilidad que otras.

```
$ pip install virtualenvwrapper

$ export WORKON_HOME=~/Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
$ printf '\n%s\n%s\n%s' '# virtualenv' 'export WORKON_HOME=~/virtualenvs' 'source
/home/salayhin/bin/virtualenvwrapper.sh' >> ~/.bashrc
$ source ~/.bashrc

$ mkvirtualenv python_3.5
Installing
setuptools.....................................
...............................................
...............................................
```

```
..............................done.
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postactivate New
python executable in python_3.5/bin/python

(python_3.5)$ ls $WORKON_HOME
python_3.5 hook.log
```

Ahora podemos instalar algún software en el entorno.

```
(python_3.5)$ pip install django
Downloading/unpacking django
Downloading Django-1.1.1.tar.gz (5.6Mb): 5.6Mb downloaded
Running setup.py egg_info for package django
Installing collected packages: django
Running setup.py install for django
changing mode of build/scripts-2.6/django-admin.py from 644 to 755
changing mode of /Users/salayhin/Envs/env1/bin/django-admin.py to 755
Successfully installed django
```

Podemos ver el nuevo paquete con lssitepackages:

```
(python_3.5)$ lssitepackages
Django-1.1.1-py2.6.egg-info easy-install.pth
setuptools-0.6.10-py2.6.egg pip-0.6.3-py2.6.egg
django setuptools.pth
```

Podemos crear múltiples entornos virtuales si queremos.

Cambiar entre entornos con workon:

```
(python_3.6)$ workon python_3.5
(python_3.5)$ echo $VIRTUAL_ENV
/Users/salayhin/Envs/env1
(python_3.5)$
```

Para salir del virtualenv

```
$ deactivate
```

Lea entorno virtual con virtualenvwrapper en línea:
https://riptutorial.com/es/python/topic/9983/entorno-virtual-con-virtualenvwrapper

Capítulo 69: Entornos virtuales

Introducción

Un entorno virtual es una herramienta para mantener las dependencias requeridas por diferentes proyectos en lugares separados, mediante la creación de entornos virtuales de Python para ellos. Resuelve el "Proyecto X depende de la versión 1.x, pero, el Proyecto Y necesita el problema 4.x", y mantiene el directorio global de paquetes de sitios limpio y manejable.

Esto ayuda a aislar sus entornos para diferentes proyectos entre sí y de las bibliotecas de su sistema.

Observaciones

Los entornos virtuales son lo suficientemente útiles como para ser usados en cada proyecto. En particular, los entornos virtuales le permiten:

1. Gestionar dependencias sin necesidad de acceso root.
2. Instale diferentes versiones de la misma dependencia, por ejemplo, al trabajar en diferentes proyectos con diferentes requisitos
3. Trabaja con diferentes versiones de python.

Examples

Creando y utilizando un entorno virtual.

`virtualenv` es una herramienta para construir entornos Python aislados. Este programa crea una carpeta que contiene todos los ejecutables necesarios para usar los paquetes que un proyecto de Python necesitaría.

Instalando la herramienta virtualenv

Esto solo se requiere una vez. El programa `virtualenv` puede estar disponible a través de su distribución. En las distribuciones similares a Debian, el paquete se llama `python-virtualenv` o `python3-virtualenv` .

Alternativamente, puede instalar `virtualenv` usando pip :

```
$ pip install virtualenv
```

Creando un nuevo entorno virtual.

Esto solo se requiere una vez por proyecto. Al iniciar un proyecto para el que desea aislar dependencias, puede configurar un nuevo entorno virtual para este proyecto:

```
$ virtualenv foo
```

Esto creará una carpeta `foo` contiene scripts de herramientas y una copia del propio binario de `python` . El nombre de la carpeta no es relevante. Una vez que se crea el entorno virtual, es autónomo y no requiere más manipulación con la herramienta `virtualenv` . Ahora puedes empezar a utilizar el entorno virtual.

Activando un entorno virtual existente

Para *activar* un entorno virtual, se requiere algo de shell magic, por lo que su Python es el que está dentro de `foo` lugar del sistema. Este es el propósito del archivo de `activate` , que debe fuente en su shell actual:

```
$ source foo/bin/activate
```

Los usuarios de Windows deben escribir:

```
$ foo\Scripts\activate.bat
```

Una vez que se ha activado un entorno virtual, los archivos binarios de `python` y `pip` y todos los scripts instalados por módulos de terceros son los que están dentro de `foo` . En particular, todos los módulos instalados con `pip` se implementarán en el entorno virtual, lo que permite un entorno de desarrollo contenido. La activación del entorno virtual también debe agregar un prefijo a su solicitud como se ve en los siguientes comandos.

```
# Installs 'requests' to foo only, not globally
(foo)$ pip install requests
```

Guardar y restaurar dependencias.

Para guardar los módulos que ha instalado a través de `pip` , puede enumerar todos esos módulos (y las versiones correspondientes) en un archivo de texto usando el comando `freeze` . Esto permite a otros instalar rápidamente los módulos de Python necesarios para la aplicación mediante el comando de instalación. El nombre convencional para tal archivo es `requirements.txt` :

```
(foo)$ pip freeze > requirements.txt
(foo)$ pip install -r requirements.txt
```

Tenga en cuenta que la `freeze` enumera todos los módulos, incluidas las dependencias transitivas requeridas por los módulos de nivel superior que instaló manualmente. Como tal, es posible que prefieren elaborar el `requirements.txt` archivo a mano , por poner sólo los módulos de alto nivel

que necesita.

Salir de un entorno virtual.

Si ha terminado de trabajar en el entorno virtual, puede desactivarlo para volver a su shell normal:

```
(foo)$ deactivate
```

Usando un entorno virtual en un host compartido

A veces no es posible `$ source bin/activate` un virtualenv, por ejemplo, si está utilizando mod_wsgi en un host compartido o si no tiene acceso a un sistema de archivos, como en Amazon API Gateway o Google AppEngine. Para esos casos, puede implementar las bibliotecas que instaló en su virtualenv local y parchear su `sys.path` .

Luckly virtualenv viene con un script que actualiza tanto su `sys.path` como su `sys.prefix`

```
import os

mydir = os.path.dirname(os.path.realpath(__file__))
activate_this = mydir + '/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

Debe agregar estas líneas al principio del archivo que ejecutará su servidor.

Encontrará el archivo `bin/activate_this.py` `virtualenv` que `virtualenv` creó en el mismo directorio que está ejecutando y agregará `lib/python2.7/site-packages` a `sys.path`

Si usted está buscando para utilizar el `activate_this.py` guión, recuerde que debe desplegar con, al menos, el `bin` y `lib/python2.7/site-packages` directorios y sus contenidos.

Python 3.x 3.3

Entornos virtuales incorporados

A partir de Python 3.3, el módulo venv creará entornos virtuales. El comando `pyvenv` no necesita instalarse por separado:

```
$ pyvenv foo
$ source foo/bin/activate
```

o

```
$ python3 -m venv foo
$ source foo/bin/activate
```

Instalación de paquetes en un entorno virtual

Una vez que se haya activado su entorno virtual, cualquier paquete que instale ahora se instalará en `virtualenv` y no de manera global. Por lo tanto, los nuevos paquetes pueden ser sin necesidad de privilegios de root.

Para verificar que los paquetes se están instalando en el `virtualenv` ejecute el siguiente comando para verificar la ruta del ejecutable que se está utilizando:

```
(<Virtualenv Name) $ which python
/<Virtualenv Directory>/bin/python

(Virtualenv Name) $ which pip
/<Virtualenv Directory>/bin/pip
```

Cualquier paquete que se instale utilizando pip se instalará en el `virtualenv` en el siguiente directorio:

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

Alternativamente, puede crear un archivo con los paquetes necesarios.

requisitos.txt :

```
requests==2.10.0
```

Ejecutando:

```
# Install packages from requirements.txt
pip install -r requirements.txt
```

Instalaré la versión 2.10.0 de las `requests` paquetes.

También puede obtener una lista de los paquetes y sus versiones instaladas actualmente en el entorno virtual activo:

```
# Get a list of installed packages
pip freeze

# Output list of packages and versions into a requirement.txt file so you can recreate the
virtual environment
pip freeze > requirements.txt
```

Alternativamente, no tiene que activar su entorno virtual cada vez que tenga que instalar un paquete. Puede usar directamente el ejecutable pip en el directorio del entorno virtual para instalar paquetes.

```
$ /<Virtualenv Directory>/bin/pip install requests
```

Puede encontrar más información sobre el uso de pip en el tema PIP .

Dado que está instalando sin root en un entorno virtual, esto *no* es una instalación global en todo el sistema: el paquete instalado solo estará disponible en el entorno virtual actual.

Creando un entorno virtual para una versión diferente de python

Suponiendo que `python` y `python3` estén instalados, es posible crear un entorno virtual para Python 3, incluso si `python3` no es el Python predeterminado:

```
virtualenv -p python3 foo
```

o

```
virtualenv --python=python3 foo
```

o

```
python3 -m venv foo
```

o

```
pyvenv foo
```

En realidad, puede crear un entorno virtual basado en cualquier versión de Python en funcionamiento de su sistema. Puede consultar diferentes python de trabajo en `/usr/bin/` o `/usr/local/bin/` (En Linux) O en `/Library/Frameworks/Python.framework/Versions/XX/bin/` (OSX), luego descifre la `--python` y usa eso en la `--python` o `-p` al crear un entorno virtual.

Gestionar múltiples entornos virtuales con virtualenvwrapper

La utilidad `virtualenvwrapper` simplifica el trabajo con entornos virtuales y es especialmente útil si está tratando con muchos proyectos / entornos virtuales.

En lugar de tener que lidiar con los directorios del entorno virtual, `virtualenvwrapper` administra por usted, almacenando todos los entornos virtuales en un directorio central (`~/.virtualenvs` por defecto).

Instalación

Instale `virtualenvwrapper` con el administrador de paquetes de su sistema.

Basado en Debian / Ubuntu:

```
apt-get install virtualenvwrapper
```

Fedora / CentOS / RHEL:

```
yum install python-virtualenvrwapper
```

Arch Linux:

```
pacman -S python-virtualenvwrapper
```

O instálalo desde PyPI usando `pip` :

```
pip install virtualenvwrapper
```

Bajo Windows, puedes usar `virtualenvwrapper-win` o `virtualenvwrapper-powershell` en `virtualenvwrapper-powershell` lugar.

Uso

Los entornos virtuales se crean con `mkvirtualenv` . Todos los argumentos del comando `virtualenv` original también son aceptados.

```
mkvirtualenv my-project
```

o por ejemplo

```
mkvirtualenv --system-site-packages my-project
```

El nuevo entorno virtual se activa automáticamente. En nuevos shells puede habilitar el entorno virtual con `workon`

```
workon my-project
```

La ventaja del comando `workon` comparado con el tradicional . `path/to/my-env/bin/activate` es que el comando `workon` funcionará en cualquier directorio; no tiene que recordar en qué directorio está almacenado el entorno virtual particular de su proyecto.

Directorios de proyectos

Incluso puede especificar un directorio de proyecto durante la creación del entorno virtual con la opción `-a` o más adelante con el comando `setvirtualenvproject` .

```
mkvirtualenv -a /path/to/my-project my-project
```

o

```
workon my-project
cd /path/to/my-project
```

```
setvirtualenvproject
```

La configuración de un proyecto hará que el comando `workon` cambie al proyecto automáticamente y habilite el comando `cdproject` que le permite cambiar al directorio del proyecto.

Para ver una lista de todos los virtualenvs gestionados por virtualenvwrapper, use `lsvirtualenv` .

Para eliminar un virtualenv, use `rmvirtualenv` :

```
rmvirtualenv my-project
```

Cada virtualenv administrado por virtualenvwrapper incluye 4 scripts de bash vacíos: `preactivate` , `postactivate` , `predeactivate` y `postdeactivate` . Estos sirven como enlaces para ejecutar comandos bash en ciertos puntos del ciclo de vida de virtualenv; por ejemplo, cualquier comando en el script `postactivate` se ejecutará justo después de que se active virtualenv. Este sería un buen lugar para establecer variables de entorno especiales, alias o cualquier otra cosa relevante. Los 4 scripts se encuentran en `.virtualenvs/<virtualenv_name>/bin/` .

Para más detalles lea la documentación de virtualenvwrapper .

Descubrir qué entorno virtual está utilizando

Si está utilizando el indicador de `bash` predeterminado en Linux, debería ver el nombre del entorno virtual al inicio de su indicador.

```
(my-project-env) user@hostname:~$ which python
/home/user/my-project-env/bin/python
```

Especificando la versión específica de Python para usar en el script en Unix / Linux

Para especificar qué versión de python el shell de Linux debe usar, la primera línea de scripts de Python puede ser una línea shebang, que comienza con `#!` :

```
#!/usr/bin/python
```

Si está en un entorno virtual, entonces `python myscript.py` Python usará Python de su entorno virtual, pero `./myscript.py` usará el intérprete de Python en el `#!` línea. Para asegurarse de que se utiliza Python del entorno virtual, cambie la primera línea a:

```
#!/usr/bin/env python
```

Después de especificar la línea shebang, recuerde dar permisos de ejecución al script haciendo lo siguiente:

```
chmod +x myscript.py
```

Hacer esto le permitirá ejecutar el script ejecutando `./myscript.py` (o proporcionará la ruta absoluta al script) en lugar de `python myscript.py` o `python3 myscript.py` .

Usando virtualenv con cáscara de pescado

Fish Shell es más amigable, pero es posible que tengas problemas al usar `virtualenv` o `virtualenvwrapper` . Alternativamente existe el `virtualfish` para el rescate. Simplemente siga la secuencia a continuación para comenzar a usar Fish Shell con virtualenv.

- Instalar virtualfish en el espacio global

```
sudo pip install virtualfish
```

- Cargue el módulo virtual de python durante el inicio de shell de peces.

```
$ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
```

- Edite esta función `fish_prompt` con `$ funced fish_prompt --editor vim` y agregue las líneas siguientes y cierre el editor vim

```
 if set -q VIRTUAL_ENV
     echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color
normal) " "
 end
```

  Nota: Si no está familiarizado con vim, simplemente suministre a su editor favorito como este `$ funced fish_prompt --editor nano` o `$ funced fish_prompt --editor gedit`

- Guardar cambios utilizando `funcsave`

```
funcsave fish_prompt
```

- Para crear un nuevo entorno virtual usa `vf new`

```
vf new my_new_env # Make sure $HOME/.virtualenv exists
```

- Si desea crear un nuevo entorno python3, especifíquelo mediante el indicador `-p`

```
vf new -p python3 my_new_env
```

- Para cambiar entre entornos virtuales, use `vf deactivate` y `vf activate another_env`

Enlaces oficiales:

- https://github.com/adambrenecki/virtualfish
- http://virtualfish.readthedocs.io/en/latest/

Realización de entornos virtuales utilizando Anaconda.

Una poderosa alternativa a `virtualenv` es Anaconda : un administrador de paquetes multiplataforma, similar a `pip` , con características para crear y eliminar rápidamente entornos virtuales. Después de instalar Anaconda, aquí hay algunos comandos para comenzar:

Crear un entorno

```
conda create --name <envname> python=<version>
```

donde `<envname>` en un nombre arbitrario para su entorno virtual, y `<version>` es una versión específica de Python que desea configurar.

Activa y desactiva tu entorno.

```
# Linux, Mac
source activate <envname>
source deactivate
```

o

```
# Windows
activate <envname>
deactivate
```

Ver una lista de entornos creados.

```
conda env list
```

Eliminar un entorno

```
conda env remove -n <envname>
```

Encuentra más comandos y características en la documentación oficial de conda .

Comprobando si se ejecuta dentro de un entorno virtual

A veces, el indicador de comandos de la shell no muestra el nombre del entorno virtual y quiere estar seguro de si está en un entorno virtual o no.

Ejecuta el intérprete de python y prueba:

```
import sys
sys.prefix
sys.real_prefix
```

- Fuera de un entorno virtual, `sys.prefix` apuntará a la instalación de python del sistema y `sys.real_prefix` no está definido.

- Dentro de un entorno virtual, `sys.prefix` apuntará a la instalación de python del entorno virtual y `sys.real_prefix` apuntará a la instalación de python del sistema.

Para los entornos virtuales creados con el módulo venv de la biblioteca estándar , no hay `sys.real_prefix` . En su lugar, compruebe si `sys.base_prefix` es el mismo que `sys.prefix` .

Lea Entornos virtuales en línea: https://riptutorial.com/es/python/topic/868/entornos-virtuales

Capítulo 70: Entrada y salida básica

Examples

Usando input () y raw_input ()

Python 2.x 2.3

`raw_input` esperará a que el usuario ingrese texto y luego devuelva el resultado como una cadena.

```
foo = raw_input("Put a message here that asks the user for input")
```

En el ejemplo anterior, `foo` almacenará cualquier entrada que proporcione el usuario.

Python 3.x 3.0

`input` esperará a que el usuario ingrese texto y luego devuelva el resultado como una cadena.

```
foo = input("Put a message here that asks the user for input")
```

En el ejemplo anterior, `foo` almacenará cualquier entrada que proporcione el usuario.

Usando la función de impresión

Python 3.x 3.0

En Python 3, la funcionalidad de impresión tiene la forma de una función:

```
print("This string will be displayed in the output")
# This string will be displayed in the output

print("You can print \n escape characters too.")
# You can print escape characters too.
```

Python 2.x 2.3

En Python 2, la impresión fue originalmente una declaración, como se muestra a continuación.

```
print "This string will be displayed in the output"
# This string will be displayed in the output

print "You can print \n escape characters too."
# You can print escape characters too.
```

Nota: el uso `from __future__ import print_function` en Python 2 permitirá a los usuarios usar la función `print()` igual que el código de Python 3. Esto solo está disponible en Python 2.6 y superior.

Función para pedir al usuario un número

```
def input_number(msg, err_msg=None):
    while True:
        try:
            return float(raw_input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)


def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)
```

Y para usarlo:

```
user_number = input_number("input a number: ", "that's not a number!")
```

O, si no desea un "mensaje de error":

```
user_number = input_number("input a number: ")
```

Imprimir una cadena sin una nueva línea al final

Python 2.x 2.3

En Python 2.x, para continuar una línea con la `print` , finalice la declaración de `print` con una coma. Se agregará automáticamente un espacio.

```
print "Hello,",
print "World!"
# Hello, World!
```

Python 3.x 3.0

En Python 3.x, la función de `print` tiene un parámetro `end` opcional que es lo que se imprime al final de la cadena dada. Por defecto es un carácter de nueva línea, por lo que es equivalente a esto:

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

Pero podrías pasar en otras cuerdas.

```
print("Hello, ", end="")
print("World!")
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

Si desea más control sobre la salida, puede usar `sys.stdout.write` :

```
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Leer de stdin

Los programas de Python pueden leer desde las tuberías de Unix . Aquí hay un ejemplo simple de cómo leer desde `stdin` :

```
import sys

for line in sys.stdin:
    print(line)
```

Tenga en cuenta que `sys.stdin` es una secuencia. Esto significa que el for-loop solo terminará cuando la secuencia haya finalizado.

Ahora puede canalizar la salida de otro programa a su programa python de la siguiente manera:

```
$ cat myfile | python myprogram.py
```

En este ejemplo, `cat myfile` puede ser cualquier comando de Unix que salga a `stdout` .

Alternativamente, usar el módulo fileinput puede ser útil:

```
import fileinput
for line in fileinput.input():
    process(line)
```

Entrada desde un archivo

La entrada también se puede leer desde archivos. Los archivos se pueden abrir utilizando la función incorporada `open` . Usar un `with <command> as <name>` sintaxis `with <command> as <name>` (llamado 'Administrador de contexto') hace que el uso de `open` y el manejo del archivo sea súper fácil:

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

Esto asegura que cuando la ejecución del código deja el bloque, el archivo se cierre automáticamente.

Los archivos se pueden abrir en diferentes modos. En el ejemplo anterior, el archivo se abre como de solo lectura. Para abrir un archivo existente para lectura solamente use `r` . Si quieres leer ese archivo como bytes usa `rb` . Para adjuntar datos a un archivo existente use `a` archivo. Use `w` para crear un archivo o sobrescribir cualquier archivo existente del mismo nombre. Puedes usar `r+` para abrir un archivo para leer y escribir. El primer argumento de `open()` es el nombre del archivo, el segundo es el modo. Si el modo se deja en blanco, el valor predeterminado será `r` .

```
# let's create an example file:
with open('shoppinglist.txt', 'w') as fileobj:
    fileobj.write('tomato\npasta\ngarlic')

with open('shoppinglist.txt', 'r') as fileobj:
    # this method makes a list where each line
    # of the file is an element in the list
    lines = fileobj.readlines()

print(lines)
# ['tomato\n', 'pasta\n', 'garlic']

with open('shoppinglist.txt', 'r') as fileobj:
    # here we read the whole content into one string:
    content = fileobj.read()
    # get a list of lines, just like int the previous example:
    lines = content.split('\n')

print(lines)
# ['tomato', 'pasta', 'garlic']
```

Si el tamaño del archivo es pequeño, es seguro leer todo el contenido del archivo en la memoria. Si el archivo es muy grande, a menudo es mejor leer línea por línea o por fragmentos, y procesar la entrada en el mismo bucle. Para hacer eso:

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

Al leer archivos, tenga en cuenta los caracteres de salto de línea específicos del sistema operativo. Aunque `for line in fileobj` los `for line in fileobj` automáticamente, siempre es seguro llamar a `strip()` en las líneas leídas, como se muestra arriba.

Los archivos `fileobj` (`fileobj` en los ejemplos anteriores) siempre apuntan a una ubicación específica en el archivo. Cuando se abren por primera vez, el identificador de archivo apunta al principio del archivo, que es la posición `0` . El identificador de archivo puede mostrar su posición actual con `tell` :

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

Al leer todo el contenido, la posición del manejador de archivos se señalará al final del archivo:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

La posición del manejador de archivos se puede configurar para lo que sea necesario:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

También puede leer cualquier longitud del contenido del archivo durante una llamada determinada. Para hacer esto pasa un argumento para `read()` . Cuando se llama a `read()` sin ningún argumento, se leerá hasta el final del archivo. Si pasa un argumento, leerá ese número de bytes o caracteres, dependiendo del modo (`rb` y `r` respectivamente):

```
# reads the next 4 characters
# starting at the current position
next4 = fileobj.read(4)
# what we got?
print(next4) # 'cucu'
# where we are now?
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 11, as we was at 7, and read 4 chars.

fileobj.close()
```

Para demostrar la diferencia entre caracteres y bytes:

```
with open('shoppinglist.txt', 'r') as fileobj:
    print(type(fileobj.read())) # <class 'str'>

with open('shoppinglist.txt', 'rb') as fileobj:
    print(type(fileobj.read())) # <class 'bytes'>
```

Lea Entrada y salida básica en línea: https://riptutorial.com/es/python/topic/266/entrada-y-salida-basica

Capítulo 71: Entrada, subconjunto y salida de archivos de datos externos utilizando Pandas

Introducción

Esta sección muestra el código básico para leer, sububicar y escribir archivos de datos externos utilizando pandas.

Examples

Código básico para importar, subcontratar y escribir archivos de datos externos mediante Pandas

```
# Print the working directory
import os
print os.getcwd()
# C:\Python27\Scripts

# Set the working directory
os.chdir('C:/Users/general1/Documents/simple Python files')
print os.getcwd()
# C:\Users\general1\Documents\simple Python files

# load pandas
import pandas as pd

# read a csv data file named 'small_dataset.csv' containing 4 lines and 3 variables
my_data = pd.read_csv("small_dataset.csv")
my_data
#     x   y   z
# 0   1   2   3
# 1   4   5   6
# 2   7   8   9
# 3  10  11  12

my_data.shape       # number of rows and columns in data set
# (4, 3)

my_data.shape[0]    # number of rows in data set
# 4

my_data.shape[1]    # number of columns in data set
# 3

# Python uses 0-based indexing.  The first row or column in a data set is located
# at position 0.  In R the first row or column in a data set is located
# at position 1.

# Select the first two rows
my_data[0:2]
#    x   y   z
#0   1   2   3
```

```
#1   4   5   6

# Select the second and third rows
my_data[1:3]
#    x  y  z
# 1  4  5  6
# 2  7  8  9

# Select the third row
my_data[2:3]
#    x   y   z
#2   7   8   9

# Select the first two elements of the first column
my_data.iloc[0:2, 0:1]
#    x
# 0  1
# 1  4

# Select the first element of the variables y and z
my_data.loc[0, ['y', 'z']]
# y    2
# z    3

# Select the first three elements of the variables y and z
my_data.loc[0:2, ['y', 'z']]
#    y  z
# 0  2  3
# 1  5  6
# 2  8  9

# Write the first three elements of the variables y and z
# to an external file.  Here index = 0 means do not write row names.

my_data2 = my_data.loc[0:2, ['y', 'z']]

my_data2.to_csv('my.output.csv', index = 0)
```

Lea Entrada, subconjunto y salida de archivos de datos externos utilizando Pandas en línea: https://riptutorial.com/es/python/topic/8854/entrada--subconjunto-y-salida-de-archivos-de-datos-externos-utilizando-pandas

Capítulo 72: Enumerar

Observaciones

Las enumeraciones se agregaron a Python en la versión 3.4 por PEP 435 .

Examples

Creación de una enumeración (Python 2.4 a 3.3)

Las enumeraciones se han cargado de Python 3.4 a Python 2.4 a través de Python 3.3. Puede obtener este backport enum34 desde PyPI.

```
pip install enum34
```

La creación de una enumeración es idéntica a cómo funciona en Python 3.4+

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3

print(Color.red)  # Color.red
print(Color(1))  # Color.red
print(Color['red'])  # Color.red
```

Iteración

Las enumeraciones son iterables:

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3

[c for c in Color]  # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

Lea Enumerar en línea: https://riptutorial.com/es/python/topic/947/enumerar

Capítulo 73: Errores comunes

Introducción

Python es un lenguaje destinado a ser claro y legible sin ambigüedades ni comportamientos inesperados. Desafortunadamente, estos objetivos no son alcanzables en todos los casos, y es por eso que Python tiene algunos casos de esquina en los que podría hacer algo diferente de lo que esperabas.

Esta sección le mostrará algunos problemas que puede encontrar al escribir código Python.

Examples

Cambiando la secuencia sobre la que estás iterando

Un bucle `for` repite en una secuencia, por lo que **alterar esta secuencia dentro del bucle podría generar resultados inesperados** (especialmente al agregar o eliminar elementos):

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]
```

Nota: `list.pop()` se está utilizando para eliminar elementos de la lista.

El segundo elemento no se eliminó porque la iteración pasa por los índices en orden. El bucle anterior se repite dos veces, con los siguientes resultados:

```
# Iteration #1
index = 0
alist = [0, 1, 2]
alist.pop(0) # removes '0'

# Iteration #2
index = 1
alist = [1, 2]
alist.pop(1) # removes '2'

# loop terminates, but alist is not empty:
alist = [1]
```

Este problema surge porque los índices cambian mientras se iteran en la dirección del índice creciente. Para evitar este problema, puede **recorrer el bucle hacia atrás** :

```
alist = [1,2,3,4,5,6,7]
for index, item in reversed(list(enumerate(alist))):
    # delete all even items
    if item % 2 == 0:
```

```
        alist.pop(index)
print(alist)
# Out: [1, 3, 5, 7]
```

Al recorrer el bucle que comienza al final, a medida que se eliminan (o agregan) los elementos, no afecta a los índices de los elementos que aparecen anteriormente en la lista. Por lo tanto, este ejemplo eliminará correctamente todos los elementos que sean pares de `alist` .

Un problema similar surge cuando se **insertan o agregan elementos a una lista sobre la que está iterando** , lo que puede dar lugar a un bucle infinito:

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    # break to avoid infinite loop:
    if index == 20:
        break
    alist.insert(index, 'a')
print(alist)
# Out (abbreviated): ['a', 'a', ..., 'a', 'a',  0,   1,   2]
```

Sin la condición de `break` , el bucle insertaría `'a'` siempre que la computadora no se quede sin memoria y el programa pueda continuar. En una situación como esta, generalmente se prefiere crear una nueva lista y agregar elementos a la nueva lista a medida que recorre la lista original.

Cuando se utiliza un bucle `for` , **no puede modificar los elementos de la lista con la variable de marcador de posición** :

```
alist = [1,2,3,4]
for item in alist:
    if item % 2 == 0:
        item = 'even'
print(alist)
# Out: [1,2,3,4]
```

En el ejemplo anterior, **cambiar el `item` no cambia realmente nada en la lista original** . Debe usar el índice de `alist[2]` (`alist[2]`), y `enumerate()` funciona bien para esto:

```
alist = [1,2,3,4]
for index, item in enumerate(alist):
    if item % 2 == 0:
        alist[index] = 'even'
print(alist)
# Out: [1, 'even', 3, 'even']
```

Un **`while` de bucle** podría ser una mejor elección en algunos casos:

Si va a **eliminar todos los elementos** de la lista:

```
zlist = [0, 1, 2]
while zlist:
```

```
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
#      2
# After: zlist = []
```

Aunque simplemente restablecer `zlist` logrará el mismo resultado;

```
zlist = []
```

El ejemplo anterior también se puede combinar con `len()` para detenerse después de un cierto punto, o para eliminar todos los elementos excepto `x` en la lista:

```
zlist = [0, 1, 2]
x = 1
while len(zlist) > x:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
# After: zlist = [2]
```

O para **recorrer una lista mientras elimina elementos que cumplen una determinada condición** (en este caso, eliminar todos los elementos pares):

```
zlist = [1,2,3,4,5]
i = 0
while i < len(zlist):
    if zlist[i] % 2 == 0:
        zlist.pop(i)
    else:
        i += 1
print(zlist)
# Out: [1, 3, 5]
```

Observe que no incrementa `i` después de eliminar un elemento. Al eliminar el elemento en `zlist[i]` , el índice del siguiente elemento ha disminuido en uno, por lo que al marcar `zlist[i]` con el mismo valor para `i` en la siguiente iteración, estará verificando correctamente el siguiente elemento en la lista .

Una forma contraria de pensar en eliminar elementos no deseados de una lista, es **agregar elementos deseados a una nueva lista** . El ejemplo siguiente es una alternativa a este último `while` ejemplo de bucle:

```
zlist = [1,2,3,4,5]

z_temp = []
```

```
for item in zlist:
    if item % 2 != 0:
        z_temp.append(item)
zlist = z_temp
print(zlist)
# Out: [1, 3, 5]
```

Aquí estamos canalizando los resultados deseados en una nueva lista. De manera opcional, podemos reasignar la lista temporal a la variable original.

Con esta tendencia de pensamiento, puede invocar una de las funciones más elegantes y potentes de Python, **listas de comprensión** , que elimina las listas temporales y se desvía de la ideología de mutación de listas / índices in situ anteriormente discutida.

```
zlist = [1,2,3,4,5]
[item for item in zlist if item % 2 != 0]
# Out: [1, 3, 5]
```

Argumento predeterminado mutable

```
def foo(li=[]):
    li.append(1)
    print(li)

foo([2])
# Out: [2, 1]
foo([3])
# Out: [3, 1]
```

Este código se comporta como se espera, pero ¿y si no pasamos un argumento?

```
foo()
# Out: [1] As expected...

foo()
# Out: [1, 1]  Not as expected...
```

Esto se debe a que los argumentos predeterminados de las funciones y los métodos se evalúan en el momento de la **definición** en lugar del tiempo de ejecución. Así que solo tenemos una sola instancia de la lista `li` .

La forma de evitarlo es usar solo tipos inmutables para los argumentos predeterminados:

```
def foo(li=None):
    if not li:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]

foo()
```

```
# Out: [1]
```

Si bien es una mejora y, `if not li` se evalúa correctamente como `False` , muchos otros objetos también lo hacen, como las secuencias de longitud cero. Los siguientes argumentos de ejemplo pueden causar resultados no deseados:

```
x = []
foo(li=x)
# Out: [1]

foo(li="")
# Out: [1]

foo(li=0)
# Out: [1]
```

El enfoque idiomático es verificar directamente el argumento en contra del objeto `None` :

```
def foo(li=None):
    if li is None:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]
```

Lista de multiplicación y referencias comunes.

Considere el caso de crear una estructura de lista anidada multiplicando:

```
li = [[]] * 3
print(li)
# Out: [[], [], []]
```

A primera vista, podríamos pensar que tenemos una lista que contiene 3 listas anidadas diferentes. Intentemos adjuntar `1` al primero:

```
li[0].append(1)
print(li)
# Out: [[1], [1], [1]]
```

`1` obtuve adjunta a todas las listas de `li` .

La razón es que `[[]] * 3` no crea una `list` de 3 `list` diferentes. Más bien, crea una `list` contiene 3 referencias al mismo objeto de `list` . Como tal, cuando agregamos a `li[0]` el cambio es visible en todos los subelementos de `li` . Esto es equivalente a:

```
li = []
element = [[]]
li = element + element + element
print(li)
```

```
# Out: [[], [], []]
element.append(1)
print(li)
# Out: [[1], [1], [1]]
```

Esto se puede corroborar aún más si imprimimos las direcciones de memoria de la `list` contenida usando `id` :

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# Out: [6830760, 6830760, 6830760]
```

La solución es crear las listas internas con un bucle:

```
li = [[] for _ in range(3)]
```

En lugar de crear una `list` única y luego hacer 3 referencias a ella, ahora creamos 3 listas distintas diferentes. Esto, de nuevo, puede verificarse usando la función `id` :

```
print([id(inner_list) for inner_list in li])
# Out: [6331048, 6331528, 6331488]
```

También puedes hacer esto. Hace que se cree una nueva lista vacía en cada llamada `append` .

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> for k in li: print(id(k))
...
4315469256
4315564552
4315564808
```

No utilice el índice para recorrer una secuencia.

No hagas

```
for i in range(len(tab)):
    print(tab[i])
```

Hacer

```
for elem in tab:
    print(elem)
```

`for` automatizará la mayoría de las operaciones de iteración para usted.

Use enumerar si realmente necesita tanto el índice como el elemento .

```
for i, elem in enumerate(tab):
```

```
    print((i, elem))
```

Tenga cuidado al usar "==" para verificar si es verdadero o falso

```
if (var == True):
    # this will execute if var is True or 1, 1.0, 1L

if (var != True):
    # this will execute if var is neither True nor 1

if (var == False):
    # this will execute if var is False or 0 (or 0.0, 0L, 0j)

if (var == None):
    # only execute if var is None

if var:
    # execute if var is a non-empty string/list/dictionary/tuple, non-0, etc

if not var:
    # execute if var is "", {}, [], (), 0, None, etc.

if var is True:
    # only execute if var is boolean True, not 1

if var is False:
    # only execute if var is boolean False, not 0

if var is None:
    # same as var == None
```

No marque si puede, solo hágalo y maneje el error

Los pitonistas usualmente dicen "es más fácil pedir perdón que permiso".

No hagas

```
if os.path.isfile(file_path):
    file = open(file_path)
else:
    # do something
```

Hacer:

```
try:
    file = open(file_path)
except OSError as e:
    # do something
```

O incluso mejor con `Python 2.6+` :

```
with open(file_path) as file:
```

Es mucho mejor porque es mucho más genérico. Puede aplicar `try/except` a casi cualquier cosa.

No necesita preocuparse por lo que debe hacer para evitarlo, solo debe preocuparse por el error que está arriesgando.

No comprobar contra tipo

Python se escribe dinámicamente, por lo tanto, verificar el tipo hace que pierdas flexibilidad. En su lugar, utilice la escritura de pato comprobando el comportamiento. Si espera una cadena en una función, use `str()` para convertir cualquier objeto en una cadena. Si espera una lista, use `list()` para convertir cualquier iterable en una lista.

No hagas

```
def foo(name):
    if isinstance(name, str):
        print(name.lower())

def bar(listing):
    if isinstance(listing, list):
        listing.extend((1, 2, 3))
        return ", ".join(listing)
```

Hacer:

```
def foo(name) :
    print(str(name).lower())

def bar(listing) :
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

Usando la última forma, `foo` aceptará cualquier objeto. `bar` aceptará cadenas, tuplas, conjuntos, listas y mucho más. Barato SECO.

No mezclar espacios y pestañas

Usa el *objeto* como primer padre

Esto es complicado, pero te morderá a medida que tu programa crezca. Hay clases antiguas y nuevas en `Python 2.x` Los viejos son, bueno, viejos. Carecen de algunas características, y pueden tener un comportamiento incómodo con la herencia. Para ser utilizable, cualquiera de su clase debe ser del "nuevo estilo". Para ello, hazlo heredar del `object` .

No hagas

```
class Father:
    pass

class Child(Father):
    pass
```

Hacer:

```
class Father(object):
    pass


class Child(Father):
    pass
```

En `Python 3.x` todas las clases son de estilo nuevo, por lo que no necesitas hacerlo.

No inicialice los atributos de clase fuera del método init

A las personas que vienen de otros idiomas les resulta tentador porque eso es lo que haces en Java o PHP. Escribe el nombre de la clase, luego enumera sus atributos y les asigna un valor predeterminado. Parece funcionar en Python, sin embargo, esto no funciona como piensas. Al hacerlo, se configurarán los atributos de clase (atributos estáticos), luego, cuando intente obtener el atributo de objeto, le dará su valor a menos que esté vacío. En ese caso devolverá los atributos de la clase. Implica dos grandes peligros:

- Si se cambia el atributo de clase, entonces se cambia el valor inicial.
- Si configura un objeto mutable como un valor predeterminado, obtendrá el mismo objeto compartido en todas las instancias.

No (a menos que quieras estática):

```
class Car(object):
    color = "red"
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Hacer

```
class Car(object):
    def __init__(self):
        self.color = "red"
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Identidad entera y de cadena

Python utiliza el almacenamiento en caché interno para un rango de enteros para reducir la sobrecarga innecesaria de su creación repetida.

En efecto, esto puede llevar a un comportamiento confuso al comparar identidades enteras:

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
True
```

y, usando otro ejemplo:

```
>>> (255 + 1) is (255 + 1)
```

```
True
>>> (256 + 1) is (256 + 1)
False
```

¿Esperar lo?

Podemos ver que la operación de identidad `is` `True` para algunos enteros (`-3` , `256`) pero no para otros (`-8` , `257`).

Para ser más específicos, los enteros en el rango `[-5, 256]` se almacenan en caché internamente durante el inicio del intérprete y solo se crean una vez. Como tales, son **idénticos** y la comparación de sus identidades con `is` rinde `True` ; los enteros fuera de este rango son (generalmente) creados sobre la marcha y sus identidades se comparan con `False` .

Este es un error común ya que este es un rango común para las pruebas, pero a menudo, el código falla en el proceso posterior de estadificación (o peor aún, en la producción) sin una razón aparente después de funcionar perfectamente en el desarrollo.

La solución es **comparar siempre los valores utilizando el** operador de **igualdad** (`==`) y **no** el operador de identidad (`is`).

Python también mantiene referencias a los usados comúnmente cuerdas y pueden resultar en un comportamiento de manera similar confuso cuando la comparación de identidades (es decir, utilizando `is`) de cadenas.

```
>>> 'python' is 'py' + 'thon'
True
```

La cadena `'python'` se usa comúnmente, por lo que Python tiene un objeto que usan todas las referencias a la cadena `'python'` .

Para cadenas poco comunes, la comparación de identidad falla incluso cuando las cadenas son iguales.

```
>>> 'this is not a common string' is 'this is not' + ' a common string'
False
>>> 'this is not a common string' == 'this is not' + ' a common string'
True
```

Entonces, al igual que la regla para enteros, **siempre compare valores de cadena utilizando el** operador de **igualdad** (`==`) y **no** el operador de identidad (`is`).

Accediendo a los atributos de int literals.

Es posible que hayas oído que todo en Python es un objeto, incluso literales. Esto significa, por ejemplo, que `7` es un objeto, lo que significa que tiene atributos. Por ejemplo, uno de estos atributos es el `bit_length` . Devuelve la cantidad de bits necesarios para representar el valor al que se solicita.

```
x = 7
x.bit_length()
# Out: 3
```

Al ver que el código anterior funciona, podría pensar intuitivamente que `7.bit_length()` también funcionaría, solo para descubrir que genera un `SyntaxError` . ¿Por qué? porque el intérprete debe diferenciar entre un acceso de atributo y un número flotante (por ejemplo, `7.2` o `7.bit_length()`). No puede, y es por eso que se levanta una excepción.

Hay algunas formas de acceder a los atributos de los literales `int` :

```
# parenthesis
(7).bit_length()
# a space
7 .bit_length()
```

El uso de dos puntos (como este `7..bit_length()`) no funciona en este caso, porque crea un literal `float` y los flotantes no tienen el método `bit_length()` .

Este problema no existe al acceder a los atributos de los literales `float` , ya que el intérprete es lo suficientemente "inteligente" para saber que un literal `float` no puede contener dos `.` , por ejemplo:

```
7.2.as_integer_ratio()
# Out: (8106479329266893, 1125899906842624)
```

Encadenamiento de u operador

Al probar para cualquiera de varias comparaciones de igualdad:

```
if a == 3 or b == 3 or c == 3:
```

es tentador abreviar esto a

```
if a or b or c == 3: # Wrong
```

Esto está mal; el operador `or` tiene una prioridad menor que `==` , por lo que la expresión se evaluará como `if (a) or (b) or (c == 3):` La forma correcta es verificando explícitamente todas las condiciones:

```
if a == 3 or b == 3 or c == 3:  # Right Way
```

Alternativamente, la función `any()` incorporada se puede usar en lugar de encadenados `or` operadores:

```
if any([a == 3, b == 3, c == 3]): # Right
```

O, para hacerlo más eficiente:

```
if any(x == 3 for x in (a, b, c)): # Right
```

O, para hacerlo más corto:

```
if 3 in (a, b, c): # Right
```

Aquí, usamos el operador `in` para probar si el valor está presente en una tupla que contiene los valores con los que queremos comparar.

Del mismo modo, es incorrecto escribir

```
if a == 1 or 2 or 3:
```

que debe ser escrito como

```
if a in (1, 2, 3):
```

sys.argv [0] es el nombre del archivo que se está ejecutando

El primer elemento de `sys.argv[0]` es el nombre del archivo python que se está ejecutando. Los elementos restantes son los argumentos del script.

```
# script.py
import sys

print(sys.argv[0])
print(sys.argv)
```

```
$ python script.py
=> script.py
=> ['script.py']

$ python script.py fizz
=> script.py
=> ['script.py', 'fizz']

$ python script.py fizz buzz
=> script.py
=> ['script.py', 'fizz', 'buzz']
```

Los diccionarios están desordenados.

Puede esperar que un diccionario de Python se ordene por claves como, por ejemplo, un C ++ `std::map` , pero este no es el caso:

```
myDict = {'first': 1, 'second': 2, 'third': 3}
print(myDict)
# Out: {'first': 1, 'second': 2, 'third': 3}
```

```
print([k for k in myDict])
# Out: ['second', 'third', 'first']
```

Python no tiene ninguna clase incorporada que ordene automáticamente sus elementos por clave.

Sin embargo, si la ordenación no es obligatoria y solo desea que su diccionario recuerde el orden de inserción de sus pares clave / valor, puede usar `collections.OrderedDict` :

```
from collections import OrderedDict

oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])

print([k for k in oDict])
# Out: ['first', 'second', 'third']
```

Tenga en cuenta que inicializar un `OrderedDict` con un diccionario estándar no ordenará el diccionario por usted. Todo lo que hace esta estructura es *preservar* el orden de inserción de la clave.

La implementación de los diccionarios se cambió en Python 3.6 para mejorar su consumo de memoria. Un efecto secundario de esta nueva implementación es que también conserva el orden de los argumentos de palabras clave que se pasan a una función:

Python 3.x 3.6

```
def func(**kw): print(kw.keys())

func(a=1, b=2, c=3, d=4, e=5)
dict_keys(['a', 'b', 'c', 'd', 'e']) # expected order
```

> **Advertencia** : tenga en cuenta que " *el aspecto de preservar el orden de esta nueva implementación se considera un detalle de implementación y no se debe confiar en él* " , ya que puede cambiar en el futuro.

Bloqueo global de intérprete (GIL) y bloqueo de hilos

Se ha escrito mucho sobre GIL de Python . A veces puede causar confusión cuando se trata de aplicaciones de subprocesos múltiples (no confundir con multiproceso).

Aquí hay un ejemplo:

```
import math
from threading import Thread

def calc_fact(num):
    math.factorial(num)

num = 600000
t = Thread(target=calc_fact, daemon=True, args=[num])
print("About to calculate: {}!".format(num))
t.start()
print("Calculating...")
```

```
t.join()
print("Calculated")
```

Esperaría ver el `Calculating...` impreso inmediatamente después de comenzar el hilo, ¡queríamos que el cálculo se realizara en un nuevo hilo después de todo! Pero en realidad, usted ve que se imprime una vez que se completa el cálculo. Esto se debe a que el nuevo hilo se basa en una función C (`math.factorial`) que bloqueará la GIL mientras se ejecuta.

Hay un par de maneras de evitar esto. El primero es implementar tu función factorial en Python nativo. Esto permitirá que el hilo principal tome el control mientras estás dentro de tu bucle. El inconveniente es que esta solución será **mucho** más lenta, ya que ya no estamos utilizando la función C.

```
def calc_fact(num):
    """ A slow version of factorial in native Python """
    res = 1
    while num >= 1:
        res = res * num
        num -= 1
    return res
```

También puede `sleep` durante un período de tiempo antes de comenzar su ejecución. Nota: esto no permitirá que su programa interrumpa el cálculo que ocurre dentro de la función C, pero permitirá que su hilo principal continúe después del inicio, que es lo que puede esperar.

```
def calc_fact(num):
    sleep(0.001)
    math.factorial(num)
```

Fugas variables en listas de comprensión y para bucles.

Considere la siguiente lista de comprensión

Python 2.x 2.7

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 2
```

Esto ocurre solo en Python 2 debido a que la comprensión de la lista "filtra" la variable de control de bucle en el ámbito circundante (fuente). Este comportamiento puede provocar errores difíciles de encontrar y ***se ha corregido en Python 3*** .

Python 3.x 3.0

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 0
```

Del mismo modo, para los bucles no tienen ámbito privado para su variable de iteración

```
i = 0
for i in range(3):
    pass
print(i) # Outputs 2
```

Este tipo de comportamiento ocurre tanto en Python 2 como en Python 3.

Para evitar problemas con las variables con fugas, use nuevas variables en la lista de comprensión y para los bucles, según corresponda.

Retorno múltiple

La función xyz devuelve dos valores a y b:

```
def xyz():
  return a, b
```

El código que llama a xyz almacena el resultado en una variable, asumiendo que xyz devuelve solo un valor:

```
t = xyz()
```

Valor de `t` es en realidad una tupla (a, b) por lo que cualquier acción en `t` suponiendo que no es una tupla puede fallar **profundo** en el código con un un **error** inesperado sobre tuplas.

> TypeError: el tipo tuple no define ... el método

La solución sería hacer:

```
a, b = xyz()
```

¡Los principiantes tendrán problemas para encontrar el motivo de este mensaje al solo leer el mensaje de error de la tupla!

Teclas JSON pitónicas

```
my_var = 'bla';
api_key = 'key';
...lots of code here...
params = {"language": "en", my_var: api_key}
```

Si estás acostumbrado a JavaScript, la evaluación de variables en los diccionarios de Python no será lo que esperas que sea. Esta declaración en JavaScript daría como resultado el objeto `params` siguiente manera:

```
{
    "language": "en",
    "my_var": "key"
}
```

En Python, sin embargo, daría como resultado el siguiente diccionario:

```
{
    "language": "en",
    "bla": "key"
}
```

`my_var` se evalúa y su valor se utiliza como clave.

Lea Errores comunes en línea: https://riptutorial.com/es/python/topic/3553/errores-comunes

Capítulo 74: Escribiendo a CSV desde String o List

Introducción

Escribir en un archivo .csv no es diferente a escribir en un archivo regular en la mayoría de los casos, y es bastante sencillo. De la mejor manera posible, cubriré el enfoque más fácil y eficiente del problema.

Parámetros

Parámetro	Detalles
abierto (**"/ ruta /"** , "modo")	Especifique la ruta a su archivo CSV
abierto (ruta, **"modo"**)	Especifique el modo para abrir el archivo en (lectura, escritura, etc.)
csv.writer (**archivo** , delimitador)	Pase el archivo CSV abierto aquí
csv.writer (archivo, **delimitador = "**)	Especificar carácter o patrón delimitador

Observaciones

```
open( path, "wb")
```

`"wb"` - Modo de escritura.

El parámetro `b` en `"wb"` que hemos utilizado, es necesario solo si desea abrirlo en modo binario, que solo se necesita en algunos sistemas operativos como Windows.

```
csv.writer ( csv_file, delimiter=',' )
```

Aquí el delimitador que hemos utilizado es `,` porque queremos que cada celda de datos en una fila, contenga el nombre, el apellido y la edad respectivamente. Ya que nuestra lista está dividida a lo largo de `,` también, resulta bastante conveniente para nosotros.

Examples

Ejemplo básico de escritura

```
import csv
```

```
#------ We will write to CSV in this function ------------

def csv_writer(data, path):

    #Open CSV file whose path we passed.
    with open(path, "wb") as csv_file:

        writer = csv.writer(csv_file, delimiter=',')
        for line in data:
            writer.writerow(line)



#---- Define our list here, and call function ------------

if __name__ == "__main__":

    """
    data = our list that we want to write.
    Split it so we get a list of lists.
    """
    data = ["first_name,last_name,age".split(","),
            "John,Doe,22".split(","),
            "Jane,Doe,31".split(","),
            "Jack,Reacher,27".split(",")
            ]

    # Path to CSV file we want to write to.
    path = "output.csv"
    csv_writer(data, path)
```

Anexando una cadena como nueva línea en un archivo CSV

```
def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "\n")
```

Lea Escribiendo a CSV desde String o List en línea:
https://riptutorial.com/es/python/topic/10862/escribiendo-a-csv-desde-string-o-list

Capítulo 75: Eventos enviados de Python Server

Introducción

Server Sent Events (SSE) es una conexión unidireccional entre un servidor y un cliente (generalmente un navegador web) que permite al servidor "enviar" información al cliente. Se parece mucho a los websockets y las encuestas largas. La principal diferencia entre SSE y websockets es que SSE es unidireccional, solo el servidor puede enviar información al cliente, mientras que al igual que con websockets, ambos pueden enviarse información entre ellos. Generalmente, se considera que SSE es mucho más simple de usar / implementar que websockets.

Examples

Frasco SSE

```
@route("/stream")
def stream():
    def event_stream():
        while True:
            if message_to_send:
                yield "data:
                    {}\n\n".format(message_to_send)"

    return Response(event_stream(), mimetype="text/event-stream")
```

Asyncio SSE

Este ejemplo utiliza la biblioteca SSE de asyncio: https://github.com/brutasse/asyncio-sse

```
import asyncio
import sse

class Handler(sse.Handler):
    @asyncio.coroutine
    def handle_request(self):
        yield from asyncio.sleep(2)
        self.send('foo')
        yield from asyncio.sleep(2)
        self.send('bar', event='wakeup')

start_server = sse.serve(Handler, 'localhost', 8888)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Lea Eventos enviados de Python Server en línea:
https://riptutorial.com/es/python/topic/9100/eventos-enviados-de-python-server

Capítulo 76: Examen de la unidad

Observaciones

Hay varias herramientas de prueba de unidad para Python. Este tema de documentación describe el módulo de `unittest` básico. Otras herramientas de prueba incluyen `py.test` y `nosetests` . Esta documentación de python sobre pruebas compara varias de estas herramientas sin profundizar.

Examples

Pruebas de excepciones

Los programas lanzan errores cuando, por ejemplo, se da una entrada incorrecta. Debido a esto, uno debe asegurarse de que se produce un error cuando se da una entrada incorrecta real. Por eso necesitamos verificar una excepción exacta, para este ejemplo usaremos la siguiente excepción:

```
class WrongInputException(Exception):
    pass
```

Esta excepción se genera cuando se proporciona una entrada incorrecta, en el siguiente contexto donde siempre esperamos un número como entrada de texto.

```
def convert2number(random_input):
    try:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Expected an integer!")
    return my_input
```

Para verificar si se ha generado una excepción, usamos `assertRaises` para verificar esa excepción. `assertRaises` se puede utilizar de dos maneras:

1. Usando la llamada de función regular. El primer argumento toma el tipo de excepción, el segundo un llamable (generalmente una función) y el resto de los argumentos se pasan a este llamable.
2. Usar una cláusula `with` , dando solo el tipo de excepción a la función. Esto tiene la ventaja de que se puede ejecutar más código, pero se debe usar con cuidado ya que múltiples funciones pueden usar la misma excepción, que puede ser problemática. Un ejemplo: con self.assertRaises (WrongInputException): convert2number ("no es un número")

Este primero se ha implementado en el siguiente caso de prueba:

```
import unittest

class ExceptionTestCase(unittest.TestCase):

```

```
    def test_wrong_input_string(self):
        self.assertRaises(WrongInputException, convert2number, "not a number")

    def test_correct_input(self):
        try:
            result = convert2number("56")
            self.assertIsInstance(result, int)
        except WrongInputException:
            self.fail()
```

También puede ser necesario verificar si hay una excepción que no debería haberse lanzado. Sin embargo, una prueba fallará automáticamente cuando se lance una excepción y, por lo tanto, puede que no sea necesaria en absoluto. Solo para mostrar las opciones, el segundo método de prueba muestra un caso sobre cómo se puede verificar que no se produzca una excepción. Básicamente, esto es capturar la excepción y luego fallar la prueba usando el método de `fail` .

Funciones de simulación con unittest.mock.create_autospec

Una forma de simular una función es utilizar la función `create_autospec` , que `create_autospec` un objeto de acuerdo con sus especificaciones. Con las funciones, podemos usar esto para asegurarnos de que se llamen adecuadamente.

Con una función `multiply` en `custom_math.py` :

```
def multiply(a, b):
    return a * b
```

Y una función `multiples_of` en `process_math.py` :

```
from custom_math import multiply


def multiples_of(integer, *args, num_multiples=0, **kwargs):
    """
    :rtype: list
    """
    multiples = []

    for x in range(1, num_multiples + 1):
        """
        Passing in args and kwargs here will only raise TypeError if values were
        passed to multiples_of function, otherwise they are ignored. This way we can
        test that multiples_of is used correctly. This is here for an illustration
        of how create_autospec works. Not recommended for production code.
        """
        multiple = multiply(integer,x, *args, **kwargs)
        multiples.append(multiple)

    return multiples
```

Podemos probar `multiples_of` solo burlándose de `multiply` . El siguiente ejemplo utiliza la prueba de unidad de la biblioteca estándar de Python, pero esto también se puede usar con otros marcos de prueba, como pytest o nose:

```
from unittest.mock import create_autospec
import unittest

# we import the entire module so we can mock out multiply
import custom_math
custom_math.multiply = create_autospec(custom_math.multiply)
from process_math import multiples_of


class TestCustomMath(unittest.TestCase):
    def test_multiples_of(self):
        multiples = multiples_of(3, num_multiples=1)
        custom_math.multiply.assert_called_with(3, 1)

    def test_multiples_of_with_bad_inputs(self):
        with self.assertRaises(TypeError) as e:
            multiples_of(1, "extra arg",  num_multiples=1) # this should raise a TypeError
```

Configuración de prueba y desmontaje dentro de un unittest.TestCase

A veces queremos preparar un contexto para cada prueba que se ejecutará. El método de `setUp` se ejecuta antes de cada prueba en la clase. `tearDown` se ejecuta al final de cada prueba. Estos métodos son opcionales. Recuerde que los TestCases a menudo se usan en herencia múltiple cooperativa, por lo que debe tener cuidado de llamar siempre `super` en estos métodos para que también se `tearDown` métodos `setUp` y `tearDown` la clase base. La implementación básica de `TestCase` proporciona `TestCase` vacíos de `setUp` y `tearDown` para que puedan llamarse sin generar excepciones:

```
import unittest


class SomeTest(unittest.TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.mock_data = [1,2,3,4,5]

    def test(self):
        self.assertEqual(len(self.mock_data), 5)

    def tearDown(self):
        super(SomeTest, self).tearDown()
        self.mock_data = []


if __name__ == '__main__':
    unittest.main()
```

Tenga en cuenta que en python2.7 +, también existe el método `addCleanup` que registra las funciones que deben llamarse después de ejecutar la prueba. A diferencia de `tearDown` que solo se llama si `setUp` tiene éxito, las funciones registradas a través de `addCleanup` se `addCleanup` incluso en el caso de una excepción no controlada en `setUp` . Como ejemplo concreto, con frecuencia se puede ver este método eliminando varios simulacros que se registraron mientras se ejecutaba la prueba:

```
import unittest
import some_module


class SomeOtherTest(unittest.TestCase):
    def setUp(self):
        super(SomeOtherTest, self).setUp()

        # Replace `some_module.method` with a `mock.Mock`
        my_patch = mock.patch.object(some_module, 'method')
        my_patch.start()

        # When the test finishes running, put the original method back.
        self.addCleanup(my_patch.stop)
```

Otra ventaja de registrar las limpiezas de esta manera es que le permite al programador colocar el código de limpieza al lado del código de configuración y lo protege en caso de que un subclase olvide llamar `super` en `tearDown` .

Afirmación de excepciones

Puede probar que una función produce una excepción con la prueba de unidad incorporada a través de dos métodos diferentes.

Usando un administrador de contexto

```
def division_function(dividend, divisor):
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError):
            x = division_function(1, 0)
```

Esto ejecutará el código dentro del administrador de contexto y, si tiene éxito, fallará la prueba porque no se generó la excepción. Si el código genera una excepción del tipo correcto, la prueba continuará.

También puede obtener el contenido de la excepción generada si desea ejecutar aserciones adicionales en su contra.

```
class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError) as ex:
            x = division_function(1, 0)

        self.assertEqual(ex.message, 'integer division or modulo by zero')
```

Al proporcionar una función de llamada

```
def division_function(dividend, divisor):
    """
```

```
    Dividing two numbers.

    :type dividend: int
    :type divisor: int

    :raises: ZeroDivisionError if divisor is zero (0).
    :rtype: int
    """
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_passing_function(self):
        self.assertRaises(ZeroDivisionError, division_function, 1, 0)
```

La excepción para verificar debe ser el primer parámetro, y una función que se puede llamar debe pasarse como el segundo parámetro. Cualquier otro parámetro especificado se pasará directamente a la función que se está llamando, lo que le permite especificar los parámetros que activan la excepción.

Eligiendo aserciones dentro de unitests

Mientras Python tiene una `assert` comunicado , el marco de la unidad de pruebas Python tiene mejores afirmaciones especializados para las pruebas: son más informativos en los fracasos, y no dependen del modo de depuración de la ejecución.

Quizás la aserción más simple es `assertTrue` , que se puede usar así:

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test_basic(self):
        self.assertTrue(1 + 1 == 2)
```

Esto funcionará bien, pero reemplazando la línea anterior con

```
        self.assertTrue(1 + 1 == 3)
```

fallará.

La afirmación `assertTrue` es bastante probable que sea la afirmación más general, ya que cualquier cosa probada puede `assertTrue` como una condición booleana, pero a menudo hay mejores alternativas. Cuando se prueba la igualdad, como arriba, es mejor escribir

```
        self.assertEqual(1 + 1, 3)
```

Cuando el primero falla, el mensaje es

```
======================================================================

FAIL: test (__main__.TruthTest)
```

```
----------------------------------------------------------------------

Traceback (most recent call last):

  File "stuff.py", line 6, in test

    self.assertTrue(1 + 1 == 3)

AssertionError: False is not true
```

pero cuando este último falla, el mensaje es

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "stuff.py", line 6, in test

    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
```

que es más informativo (en realidad evaluó el resultado del lado izquierdo).

Puede encontrar la lista de afirmaciones en la documentación estándar . En general, es una buena idea elegir la afirmación que se ajuste más específicamente a la condición. Por lo tanto, como se muestra arriba, para afirmar que `1 + 1 == 2` es mejor usar `assertEqual` que `assertTrue` . De manera similar, para afirmar que `a is None` , es mejor usar `assertIsNone` que `assertEqual` .

Tenga en cuenta también que las afirmaciones tienen formas negativas. Por `assertEqual` tanto, `assertEqual` tiene su contraparte negativa `assertNotEqual` , y `assertIsNone` tiene su contraparte negativa `assertIsNotNone` . Una vez más, usar las contrapartes negativas cuando sea apropiado, dará lugar a mensajes de error más claros.

Pruebas unitarias con pytest

instalando pytest:

```
pip install pytest
```

preparando las pruebas:

```
mkdir tests
touch tests/test_docker.py
```

Funciones para probar en `docker_something/helpers.py` :

```
from subprocess import Popen, PIPE
# this Popen is monkeypatched with the fixture `all_popens`
```

```
def copy_file_to_docker(src, dest):
    try:
        result = Popen(['docker','cp', src, 'something_cont:{}'.format(dest)], stdout=PIPE,
stderr=PIPE)
        err = result.stderr.read()
        if err:
            raise Exception(err)
    except Exception as e:
        print(e)
    return result

def docker_exec_something(something_file_string):
    fl = Popen(["docker", "exec", "-i", "something_cont", "something"], stdin=PIPE,
stdout=PIPE, stderr=PIPE)
    fl.stdin.write(something_file_string)
    fl.stdin.close()
    err = fl.stderr.read()
    fl.stderr.close()
    if err:
        print(err)
        exit()
    result = fl.stdout.read()
    print(result)
```

La prueba importa `test_docker.py` :

```
import os
from tempfile import NamedTemporaryFile
import pytest
from subprocess import Popen, PIPE

from docker_something import helpers
copy_file_to_docker = helpers.copy_file_to_docker
docker_exec_something = helpers.docker_exec_something
```

burlándose de un archivo como objeto en `test_docker.py` :

```
class MockBytes():
    '''Used to collect bytes
    '''
    all_read = []
    all_write = []
    all_close = []

    def read(self, *args, **kwargs):
        # print('read', args, kwargs, dir(self))
        self.all_read.append((self, args, kwargs))

    def write(self, *args, **kwargs):
        # print('wrote', args, kwargs)
        self.all_write.append((self, args, kwargs))

    def close(self, *args, **kwargs):
        # print('closed', self, args, kwargs)
        self.all_close.append((self, args, kwargs))

    def get_all_mock_bytes(self):
        return self.all_read, self.all_write, self.all_close
```

Parches de mono con pytest en `test_docker.py` :

```
@pytest.fixture
def all_popens(monkeypatch):
    '''This fixture overrides / mocks the builtin Popen
        and replaces stdin, stdout, stderr with a MockBytes object

        note: monkeypatch is magically imported
    '''
    all_popens = []

    class MockPopen(object):
        def __init__(self, args, stdout=None, stdin=None, stderr=None):
            all_popens.append(self)
            self.args = args
            self.byte_collection = MockBytes()
            self.stdin = self.byte_collection
            self.stdout = self.byte_collection
            self.stderr = self.byte_collection
            pass
    monkeypatch.setattr(helpers, 'Popen', MockPopen)

    return all_popens
```

Ejemplos de pruebas, deben comenzar con el prefijo `test_` en el archivo `test_docker.py` :

```
def test_docker_install():
    p = Popen(['which', 'docker'], stdout=PIPE, stderr=PIPE)
    result = p.stdout.read()
    assert 'bin/docker' in result

def test_copy_file_to_docker(all_popens):
    result = copy_file_to_docker('asdf', 'asdf')
    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert mock_read
    assert result.args == ['docker', 'cp', 'asdf', 'something_cont:asdf']


def test_docker_exec_something(all_popens):

    docker_exec_something(something_file_string)

    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert len(mock_read) == 3
    something_template_stdin = mock_write[0][1][0]
    these = [os.environ['USER'], os.environ['password_prod'], 'table_name_here', 'test_vdm',
'col_a', 'col_b', '/tmp/test.tsv']
    assert all([x in something_template_stdin for x in these])
```

ejecutando las pruebas una a la vez:

```
py.test -k test_docker_install tests
py.test -k test_copy_file_to_docker tests
py.test -k test_docker_exec_something tests
```

ejecutando todas las pruebas en la carpeta de `tests` :

```
py.test -k test_ tests
```

Lea Examen de la unidad en línea: https://riptutorial.com/es/python/topic/631/examen-de-la-unidad

Capítulo 77: Excepciones

Introducción

Los errores detectados durante la ejecución se denominan excepciones y no son incondicionalmente fatales. La mayoría de las excepciones no son manejadas por los programas; es posible escribir programas que manejen excepciones seleccionadas. Hay características específicas en Python para lidiar con las excepciones y la lógica de excepciones. Además, las excepciones tienen una jerarquía de tipos enriquecidos, todos heredados del tipo `BaseException` .

Sintaxis

- levantar *excepción*
- elevar # re-elevar una excepción que ya ha sido planteada
- generar *excepción* de *causa* # Python 3 - establecer causa de excepción
- generar *excepción* desde Ninguna # Python 3 - suprimir todo el contexto de excepción
- tratar:
- excepto *[tipos de excepción] [* como *identificador]* :
- más:
- finalmente:

Examples

Levantando excepciones

Si su código encuentra una condición que no sabe cómo manejar, como un parámetro incorrecto, debe generar la excepción apropiada.

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an odd number")

    return odds + 1
```

Atrapar excepciones

Utilice `try...except:` para detectar excepciones. Debe especificar una excepción tan precisa como pueda:

```
try:
    x = 5 / 0
except ZeroDivisionError as e:
    # `e` is the exception object
    print("Got a divide by zero! The exception was:", e)
    # handle exceptional case
    x = 0
```

```
finally:
    print "The END"
    # it runs no matter what execute.
```

La clase de excepción que se especifica, en este caso, `ZeroDivisionError` , captura cualquier excepción que sea de esa clase o de cualquier subclase de esa excepción.

Por ejemplo, `ZeroDivisionError` es una subclase de `ArithmeticError` :

```
>>> ZeroDivisionError.__bases__
(<class 'ArithmeticError'>,)
```

Y así, lo siguiente seguirá `ZeroDivisionError` :

```
try:
    5 / 0
except ArithmeticError:
    print("Got arithmetic error")
```

Ejecutando código de limpieza con finalmente

A veces, es posible que desee que ocurra algo, independientemente de la excepción que haya ocurrido, por ejemplo, si tiene que limpiar algunos recursos.

El bloque `finally` de una cláusula de `try` ocurrirá independientemente de si se produjeron excepciones.

```
resource = allocate_some_expensive_resource()
try:
    do_stuff(resource)
except SomeException as e:
    log_error(e)
    raise  # re-raise the error
finally:
    free_expensive_resource(resource)
```

Este patrón a menudo se maneja mejor con los administradores de contexto (usando la declaración `with`).

Re-elevando excepciones

A veces desea capturar una excepción solo para inspeccionarla, por ejemplo, para fines de registro. Después de la inspección, desea que la excepción continúe propagándose como lo hizo antes.

En este caso, simplemente use la instrucción `raise` sin parámetros.

```
try:
    5 / 0
except ZeroDivisionError:
    print("Got an error")
```

```
    raise
```

Sin embargo, tenga en cuenta que alguien que se encuentre más arriba en la pila de personas que llaman todavía puede detectar la excepción y manejarla de alguna manera. La salida realizada podría ser una molestia en este caso porque ocurrirá en cualquier caso (capturado o no capturado). Por lo tanto, podría ser una mejor idea plantear una excepción diferente, que contenga su comentario sobre la situación, así como la excepción original:

```
try:
    5 / 0
except ZeroDivisionError as e:
    raise ZeroDivisionError("Got an error", e)
```

Pero esto tiene el inconveniente de reducir la traza de excepción exactamente a este `raise` mientras que la `raise` sin argumento conserva la traza de excepción original.

En Python 3 puedes mantener la pila original usando la sintaxis de `raise` - `from` :

```
    raise ZeroDivisionError("Got an error") from e
```

Cadena de excepciones con aumento de

En el proceso de manejar una excepción, es posible que desee plantear otra excepción. Por ejemplo, si obtiene un `IOError` mientras lee un archivo, es posible que desee plantear un error específico de la aplicación para que se presente a los usuarios de su biblioteca.

Python 3.x 3.0

Puede encadenar excepciones para mostrar cómo procedió el manejo de las excepciones:

```
>>> try:
    5 / 0
except ZeroDivisionError as e:
    raise ValueError("Division failed") from e

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: Division failed
```

Jerarquía de excepciones

El manejo de excepciones se produce en función de una jerarquía de excepciones, determinada por la estructura de herencia de las clases de excepciones.

Por ejemplo, `IOError` y `OSError` son subclases de `EnvironmentError` . El código que captura un

`IOError` no detectará un `OSError` . Sin embargo, el código que atrapa un `EnvironmentError` detectará tanto `IOError` s como `OSError` s.

La jerarquía de las excepciones incorporadas:

Python 2.x 2.3

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
       +-- ImportWarning
       +-- UnicodeWarning
       +-- BytesWarning
```

Python 3.x 3.0

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
```

```
          +-- BytesWarning
          +-- ResourceWarning
```

Las excepciones son objetos también

Las excepciones son solo objetos de Python normales que heredan de la `BaseException` . Una secuencia de comandos de Python puede usar la instrucción `raise` para interrumpir la ejecución, lo que hace que Python imprima un seguimiento de pila de la pila de llamadas en ese punto y una representación de la instancia de excepción. Por ejemplo:

```
>>> def failing_function():
...     raise ValueError('Example error!')
>>> failing_function()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in failing_function
ValueError: Example error!
```

que dice que un `ValueError` con el mensaje `'Example error!'` fue planteado por nuestro `failing_function()` , que fue ejecutado en el intérprete.

El código de llamada puede elegir manejar cualquier excepción de todo tipo que una llamada pueda generar:

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Handled the error')
Handled the error
```

Puede obtener los objetos de excepción asignándolos en la parte `except...` del código de manejo de excepciones:

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Caught exception', repr(e))
Caught exception ValueError('Example error!',)
```

Puede encontrar una lista completa de las excepciones de Python incorporadas junto con sus descripciones en la Documentación de Python: https://docs.python.org/3.5/library/exceptions.html . Y aquí está la lista completa organizada jerárquicamente: Jerarquía de excepciones .

Creación de tipos de excepción personalizados

Crea una clase heredada de `Exception` :

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
```

```
except FooException:
    print("A FooException was raised.")
```

u otro tipo de excepción:

```
class NegativeError(ValueError):
      pass

def foo(x):
    # function that only accepts positive values of x
    if x < 0:
        raise NegativeError("Cannot process negative numbers")
    ...  # rest of function body
try:
    result = foo(int(input("Enter a positive integer: ")))  # raw_input in Python 2.x
except NegativeError:
    print("You entered a negative number!")
else:
    print("The result was " + str(result))
```

No atrapes todo!

Aunque a menudo es tentador atrapar todas las `Exception` :

```
try:
    very_difficult_function()
except Exception:
    # log / try to reconnect / exit gratiously
finally:
    print "The END"
    # it runs no matter what execute.
```

O incluso todo (que incluye `BaseException` y todos sus hijos, incluida la `Exception`):

```
try:
    even_more_difficult_function()
except:
    pass  # do whatever needed
```

En la mayoría de los casos es una mala práctica. Es posible que `SystemExit` más de lo previsto, como `SystemExit` , `KeyboardInterrupt` y `MemoryError` , cada uno de los cuales generalmente debe manejarse de manera diferente a los errores habituales del sistema o la lógica. También significa que no hay una comprensión clara de qué puede hacer mal el código interno y cómo recuperarse adecuadamente de esa condición. Si está detectando cada error, no sabrá qué error ocurrió o cómo solucionarlo.

Esto se conoce más comúnmente como "enmascaramiento de errores" y debe evitarse. Deje que su programa se bloquee en lugar de fallar silenciosamente o incluso peor, fallando en un nivel más profundo de ejecución. (Imagina que es un sistema transaccional)

Por lo general, estas construcciones se usan en el nivel más externo del programa, y registrarán los detalles del error para que el error se pueda corregir, o el error se pueda manejar más

específicamente.

Atrapando múltiples excepciones

Hay algunas maneras de atrapar múltiples excepciones .

La primera es creando una tupla de los tipos de excepción que desea capturar y manejar de la misma manera. Este ejemplo hará que el código ignore las excepciones `KeyError` y `AttributeError` .

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except (KeyError, AttributeError) as e:
    print("A KeyError or an AttributeError exception has been caught.")
```

Si desea manejar diferentes excepciones de diferentes maneras, puede proporcionar un bloque de excepción separado para cada tipo. En este ejemplo, todavía `KeyError` y `AttributeError` , pero manejamos las excepciones de diferentes maneras.

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except KeyError as e:
    print("A KeyError has occurred. Exception message:", e)
except AttributeError as e:
    print("An AttributeError has occurred. Exception message:", e)
```

Ejemplos prácticos de manejo de excepciones.

Entrada del usuario

Imagina que quieres que un usuario ingrese un número a través de una `input` . Desea asegurarse de que la entrada es un número. Puedes usar `try` / `except` de esto:

Python 3.x 3.0

```
while True:
    try:
        nb = int(input('Enter a number: '))
        break
    except ValueError:
        print('This is not a number, try again.')
```

Nota: Python 2.x usaría `raw_input` en `raw_input` lugar; la `input` la función existe en Python 2.x pero tiene una semántica diferente. En el ejemplo anterior, la `input` también aceptaría expresiones como `2 + 2` que se evalúan como un número.

Si la entrada no se pudo convertir en un entero, se `ValueError` un `ValueError` . Puedes atraparlo con `except` . Si se plantea no es una excepción, `break` salta fuera del bucle. Después del bucle, `nb` contiene un entero.

Los diccionarios

Imagine que está iterando sobre una lista de enteros consecutivos, como el `range(n)` , y tiene una lista de diccionarios `d` que contiene información sobre cosas que hacer cuando encuentra algunos enteros en particular, por ejemplo, *omita los* `d[i]` *siguientes* .

```
d = [{7: 3}, {25: 9}, {38: 5}]

for i in range(len(d)):
    do_stuff(i)
    try:
       dic = d[i]
       i += dic[i]
    except KeyError:
       i += 1
```

Se `KeyError` un `KeyError` cuando intente obtener un valor de un diccionario para una clave que no existe.

Más

El código en un bloque else solo se ejecutará si el código en el bloque `try` no genera excepciones. Esto es útil si tiene algún código que no desea ejecutar si se lanza una excepción, pero no quiere que se detecten excepciones lanzadas por ese código.

Por ejemplo:

```
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# Output: one
# Output: ValueError
```

Tenga en cuenta que este tipo de `else:` no se puede combinar con `if` inicia la cláusula else en un `elif` . Si usted tiene un siguiente `if` que necesita para mantenerse con sangría debajo de esa `else:` :

```
try:
    ...
except ...:
    ...
else:
    if ...:
```

```
    ...
elif ...:
    ...
else:
    ...
```

Lea Excepciones en línea: https://riptutorial.com/es/python/topic/1788/excepciones

Capítulo 78: Excepciones del Commonwealth

Introducción

Aquí en Stack Overflow a menudo vemos duplicados que hablan de los mismos errores: `"ImportError: No module named '??????'`, `SyntaxError: invalid syntax` o `NameError: name '???' is not defined`. Este es un esfuerzo para reducirlos y para tener alguna documentación con la cual enlazar.

Examples

IndentationErrors (o indentation SyntaxErrors)

En la mayoría de los otros idiomas, la sangría no es obligatoria, pero en Python (y otros idiomas: versiones anteriores de FORTRAN, Makefiles, Whitespace (lenguaje esotérico), etc.) no es el caso, lo que puede ser confuso si viene de otro idioma. si estaba copiando el código de un ejemplo a su propio, o simplemente si es nuevo.

IndentationError / SyntaxError: sangría inesperada

Esta excepción se produce cuando el nivel de sangrado aumenta sin razón.

Ejemplo

No hay razón para aumentar el nivel aquí:

Python 2.x 2.0 2.7

```
print "This line is ok"
    print "This line isn't ok"
```

Python 3.x 3.0

```
print("This line is ok")
    print("This line isn't ok")
```

Aquí hay dos errores: el último y que la sangría no coincide con ningún nivel de sangría. Sin embargo solo se muestra una:

Python 2.x 2.0 2.7

```
print "This line is ok"
 print "This line isn't ok"
```

Python 3.x 3.0

```
 print("This line is ok")
  print("This line isn't ok")
```

IndentationError / SyntaxError: unindent no coincide con ningún nivel de sangría externa

Parece que no te diste por completo.

Ejemplo

Python 2.x 2.0 2.7

```
def foo():
    print "This should be part of foo()"
   print "ERROR!"
print "This is not a part of foo()"
```

Python 3.x 3.0

```
 print("This line is ok")
  print("This line isn't ok")
```

Error Tabulación: Se esperaba un bloque tabulado

Después de dos puntos (y luego una nueva línea) el nivel de sangrado tiene que aumentar. Este error surge cuando eso no sucedió.

Ejemplo

```
if ok:
doStuff()
```

Nota : Use el `pass` palabra clave (que no hace absolutamente nada) para simplemente poner un `if` , `else` , `except` , `class` , `method` o `definition` pero no diga lo que sucederá si la condición de llamada es verdadera (pero hágalo más tarde, o en el caso de `except` : simplemente no hacer nada):

```
def foo():
    pass
```

IndentationError: uso incoherente de tabulaciones y espacios en sangría

Ejemplo

```
def foo():
    if ok:
      return "Two != Four != Tab"
        return "i dont care i do whatever i want"
```

Cómo evitar este error

No uses pestañas. Se desalienta por `PEP8` , la guía de estilo para Python.

1. Configure su editor para usar 4 **espacios** para la sangría.
2. Haga una búsqueda y reemplace para reemplazar todas las pestañas con 4 espacios.
3. Asegúrese de que su editor esté configurado para **mostrar las** pestañas en 8 espacios, para que pueda darse cuenta fácilmente de ese error y corregirlo.

Vea esta pregunta si desea aprender más.

TypeErrors

Estas excepciones se producen cuando el tipo de algún objeto debe ser diferente

TypeError: [definición / método] toma? argumentos posicionales pero? se le dio

Se llamó a una función o método con más (o menos) argumentos que los que puede aceptar.

Ejemplo

Si se dan más argumentos:

```
def foo(a): return a
foo(a,b,c,d) #And a,b,c,d are defined
```

Si se dan menos argumentos:

```
def foo(a,b,c,d): return a += b + c + d
foo(a) #And a is defined
```

Nota : si desea usar un número desconocido de argumentos, puede usar `*args` o `**kwargs` . Ver * args y ** kwargs

---

TypeError: tipo (s) de operando no compatibles para [operando]: '???' y '???'

Algunos tipos no se pueden operar juntos, dependiendo del operando.

Ejemplo

Por ejemplo: `+` se usa para concatenar y agregar, pero no puede usar ninguno de ellos para ambos tipos. Por ejemplo, al intentar crear un `set` concatenando (`+` ing) `'set1'` y `'tuple1'` el error. Código:

```
set1, tuple1 = {1,2}, (3,4)
a = set1 + tuple1
```

Algunos tipos (por ejemplo: `int` y `string`) usan ambos `+` pero para diferentes cosas:

```
b = 400 + 'foo'
```

O puede que ni siquiera se utilicen para nada:

```
c = ["a","b"] - [1,2]
```

Pero puedes, por ejemplo, agregar un `float` a un `int` :

```
d = 1 + 1.0
```

Error de tecleado: '???' El objeto no es iterable / subscriptible:

Para que un objeto sea iterable, puede tomar índices secuenciales desde cero hasta que los índices ya no sean válidos y se `IndexError` un `IndexError` (más técnicamente: debe tener un método `__iter__` que devuelve un `__iterator__` , o que defina un método `__getitem__` que sí lo mencionado anteriormente).

Ejemplo

Aquí estamos diciendo que la `bar` es el elemento cero de 1. Tonterías:

```
foo = 1
bar = foo[0]
```

Esta es una versión más discreta: En este ejemplo `for` intenta establecer `x` a `amount[0]` , el primer elemento en un iterable pero no puede porque cantidad es un int:

```
amount = 10
for x in amount: print(x)
```

Error de tecleado: '???' el objeto no es llamable

Está definiendo una variable y llamándola más tarde (como lo que hace con una función o método)

Ejemplo

```
foo = "notAFunction"
foo()
```

NameError: name '???' no está definido

Se genera cuando intenta utilizar una variable, método o función que no está inicializada (al menos no antes). En otras palabras, se genera cuando no se encuentra un nombre local o global solicitado. Es posible que haya escrito mal el nombre del objeto o se haya olvidado de `import` algo. También tal vez esté en otro ámbito. Los cubriremos con ejemplos separados.

Simplemente no está definido en ninguna parte en el código

Es posible que haya olvidado inicializarlo, especialmente si es una constante.

```
foo   # This variable is not defined
bar() # This function is not defined
```

Tal vez se define más adelante:

```
baz()

def baz():
    pass
```

O no fue `import`

```
#needs import math

def sqrt():
    x = float(input("Value: "))
    return math.sqrt(x)
```

Los alcances de Python y la Regla LEGB:

La llamada Regla de LEGB habla sobre los alcances de Python. Su nombre se basa en los diferentes ámbitos, ordenados por las prioridades correspondientes:

```
Local → Enclosed → Global → Built-in.
```

- **L** ocal: Variables no declaradas globalmente o asignadas en una función.
- **E** nclosing: Variables definidas en una función que está envuelta dentro de otra función.
- **Ondoparael:** Las variables declaradas globales, o asignados en el nivel superior de un archivo.
- **B** uilt-in: variables preasignadas en el módulo de nombres incorporado.

Como ejemplo:

```
for i in range(4):
    d = i * 2
print(d)
```

`d` es accesible porque el bucle `for` no marca un nuevo ámbito, pero si lo hiciera, tendríamos un error y su comportamiento sería similar a:

```
def noaccess():
    for i in range(4):
        d = i * 2
noaccess()
print(d)
```

Python dice `NameError: name 'd' is not defined`

Otros errores

AssertError

La `assert` declaración existe en casi todos los lenguajes de programación. Cuando tu lo hagas:

```
assert condition
```

o:

```
assert condition, message
```

Es equivalente a esto:

```
if __debug__:
    if not condition: raise AssertionError(message)
```

Las aserciones pueden incluir un mensaje opcional, y puedes deshabilitarlas cuando hayas terminado la depuración.

Nota : la **depuración de la** variable incorporada es Verdadero en circunstancias normales, Falso cuando se solicita la optimización (opción de línea de comando -O). Las asignaciones a **depuración** son ilegales. El valor de la variable incorporada se determina cuando se inicia el intérprete.

Teclado interrumpir

Se produjo un error cuando el usuario presiona la tecla de interrupción, normalmente `Ctrl` + `C` o `del` .

ZeroDivisionError

`1/0` calcular `1/0` que no está definido. Vea este ejemplo para encontrar los divisores de un número:

Python 2.x 2.0 2.7

```
div = float(raw_input("Divisors of: "))
for x in xrange(div+1): #includes the number itself and zero
    if div/x == div//x:
        print x, "is a divisor of", div
```

Python 3.x 3.0

```
div = int(input("Divisors of: "))
for x in range(div+1): #includes the number itself and zero
    if div/x == div//x:
        print(x, "is a divisor of", div)
```

`ZeroDivisionError` porque el bucle `for` asigna ese valor a `x` . En su lugar debería ser:

Python 2.x 2.0 2.7

```
div = float(raw_input("Divisors of: "))
for x in xrange(1,div+1): #includes the number itself but not zero
    if div/x == div//x:
        print x, "is a divisor of", div
```

Python 3.x 3.0

```
div = int(input("Divisors of: "))
for x in range(1,div+1): #includes the number itself but not zero
    if div/x == div//x:
        print(x, "is a divisor of", div)
```

Error de sintaxis en buen código

La gran mayoría de las veces que un SyntaxError que apunta a una línea sin interés significa que hay un problema en la línea anterior (en este ejemplo, es un paréntesis que falta):

```
def my_print():
    x = (1 + 1
    print(x)
```

Devoluciones

```
  File "<input>", line 3
    print(x)
        ^
SyntaxError: invalid syntax
```

La razón más común para este problema son paréntesis / paréntesis que no coinciden, como muestra el ejemplo.

Hay una advertencia importante para las declaraciones impresas en Python 3:

Python 3.x 3.0

```
 >>> print "hello world"
  File "<stdin>", line 1
    print "hello world"
                      ^
SyntaxError: invalid syntax
```

Debido a que la declaración de `print` fue reemplazada con la función `print()` , entonces usted quiere:

```
print("hello world")  # Note this is valid for both Py2 & Py3
```

Lea Excepciones del Commonwealth en línea:
https://riptutorial.com/es/python/topic/9300/excepciones-del-commonwealth

Capítulo 79: Explotación florestal

Examples

Introducción al registro de Python

Este módulo define funciones y clases que implementan un sistema flexible de registro de eventos para aplicaciones y bibliotecas.

El beneficio clave de tener la API de registro proporcionada por un módulo de biblioteca estándar es que todos los módulos de Python pueden participar en el registro, por lo que el registro de su aplicación puede incluir sus propios mensajes integrados con mensajes de módulos de terceros.

Entonces, comencemos:

Ejemplo de configuración directamente en el código

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('this is a %s test', 'debug')
```

Ejemplo de salida:

```
2016-07-26 18:53:55,332 root         DEBUG    this is a debug test
```

Ejemplo de configuración a través de un archivo INI

Suponiendo que el archivo se llama logging_config.ini. Más detalles sobre el formato de archivo se encuentran en la sección de configuración de registro del tutorial de registro .

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler
```

```
[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Luego use `logging.config.fileConfig()` en el código:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Ejemplo de configuración a través de un diccionario

A partir de Python 2.7, puede usar un diccionario con detalles de configuración. PEP 391 contiene una lista de los elementos obligatorios y opcionales en el diccionario de configuración.

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
        },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
              'formatter': 'f',
              'level': logging.DEBUG}
        },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
        },
)

dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Excepciones de registro

Si desea registrar excepciones, puede y debe hacer uso del método `logging.exception(msg)` :

```
>>> import logging
>>> logging.basicConfig()
>>> try:
...     raise Exception('foo')
```

```
... except:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

No pase la excepción como argumento:

Como `logging.exception(msg)` espera un `msg` arg, es un error común para aprobar la excepción en el registro de llamadas de esta manera:

```
>>> try:
...     raise Exception('foo')
... except Exception as e:
...     logging.exception(e)
...
ERROR:root:foo
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

Si bien puede parecer que esto es lo correcto al principio, en realidad es problemático debido a la forma en que las excepciones y las distintas codificaciones funcionan juntas en el módulo de registro:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception(e)
...
Traceback (most recent call last):
  File "/.../python2.7/logging/__init__.py", line 861, in emit
    msg = self.format(record)
  File "/.../python2.7/logging/__init__.py", line 734, in format
    return fmt.format(record)
  File "/.../python2.7/logging/__init__.py", line 469, in format
    s = self._fmt % record.__dict__
UnicodeEncodeError: 'ascii' codec can't encode characters in position 1-2: ordinal not in
range(128)
Logged from file <stdin>, line 4
```

Intentar registrar una excepción que contenga caracteres Unicode, de esta manera fallará miserablemente . Ocultará el seguimiento de pila de la excepción original al anularla con una nueva que se `logging.exception(e)` durante el formateo de su `logging.exception(e)` .

Obviamente, en su propio código, podría estar al tanto de la codificación en las excepciones. Sin embargo, las bibliotecas de terceros pueden manejar esto de una manera diferente.

Uso Correcto:

Si en lugar de la excepción simplemente pasa un mensaje y deja que Python haga su magia, funcionará:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

Como puede ver, en realidad no usamos `e` en ese caso, la llamada a `logging.exception(...)` formatea mágicamente la excepción más reciente.

Registro de excepciones con niveles de registro que no sean ERROR

Si desea registrar una excepción con un nivel de registro diferente al ERROR, puede usar el argumento `exc_info` de los registradores predeterminados:

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

Accediendo al mensaje de la excepción.

Tenga en cuenta que las bibliotecas podrían generar excepciones con los mensajes como Unicode o (utf-8 si tiene suerte) de cadenas de bytes. Si realmente necesita acceder a un texto de excepción, la única forma confiable, que siempre funcionará, es usar `repr(e)` o el formato de cadena `%r` :

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('received this exception: %r' % e)
...
ERROR:root:received this exception: Exception(u'f\xf6\xf6',)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

Lea Explotación florestal en línea: https://riptutorial.com/es/python/topic/4081/explotacion-florestal

Capítulo 80: Exposicióncción

Sintaxis

- valor1 ** valor2
- pow (valor1, valor2 [, valor3])
- value1 .__ pow __ (value2 [, value3])
- valor2 .__ rpow __ (valor1)
- operator.pow (valor1, valor2)
- operador .__ pow __ (valor1, valor2)
- math.pow (value1, value2)
- math.sqrt (valor1)
- math.exp (value1)
- cmath.exp (valor1)
- math.expm1 (value1)

Examples

Raíz cuadrada: math.sqrt () y cmath.sqrt

El módulo `math` contiene la función `math.sqrt()` que puede calcular la raíz cuadrada de cualquier número (que se puede convertir en un `float`) y el resultado siempre será un `float` :

```
import math

math.sqrt(9)                 # 3.0
math.sqrt(11.11)             # 3.3331666624997918
math.sqrt(Decimal('6.25'))   # 2.5
```

La función `math.sqrt()` genera un `ValueError` si el resultado sería `complex` :

```
math.sqrt(-10)
```

    ValueError: error de dominio matemático

`math.sqrt(x)` es *más rápido* que `math.pow(x, 0.5)` o `x ** 0.5` pero la precisión de los resultados es la misma. El módulo `cmath` es extremadamente similar al módulo `math` , excepto por el hecho de que puede calcular números complejos y todos sus resultados tienen la forma de a + bi. También puede usar `.sqrt()` :

```
import cmath

cmath.sqrt(4)  # 2+0j
cmath.sqrt(-4) # 2j
```

¿Qué pasa con la `j` ? `j` es el equivalente a la raíz cuadrada de -1. Todos los números se pueden

poner en la forma a + bi, o en este caso, a + bj. `a` es la parte real del número como el 2 en `2+0j` . Como no tiene una parte imaginaria, `b` es 0. `b` representa parte de la parte imaginaria del número como el 2 en `2j` . Como no hay una parte real en esto, `2j` también puede escribirse como `0 + 2j` .

Exposiciónción utilizando builtins: ** y pow ()

Exponenciación se puede utilizar mediante el uso de la orden interna `pow` -Función o el `**` operador:

```
2 ** 3    # 8
pow(2, 3) # 8
```

Para la mayoría de las operaciones aritméticas (todas en Python 2.x), el tipo de resultado será el del operando más amplio. Esto no es cierto para `**` ; Los siguientes casos son excepciones a esta regla:

- Base: `int` , exponente: `int < 0` :

  ```
  2 ** -3
  # Out: 0.125 (result is a float)
  ```

- Esto también es válido para Python 3.x.
- Antes de Python 2.2.0, esto `ValueError` un `ValueError` .
- Base: `int < 0` o `float < 0` , exponente: `float != int`

  ```
  (-2) ** (0.5)  # also (-2.) ** (0.5)
  # Out: (8.659560562354934e-17+1.4142135623730951j) (result is complex)
  ```

- Antes de Python 3.0.0, esto `ValueError` un `ValueError` .

El módulo `operator` contiene dos funciones que son equivalentes al operador `**` :

```
import operator
operator.pow(4, 2)       # 16
operator.__pow__(4, 3)   # 64
```

o uno podría llamar directamente al método `__pow__` :

```
val1, val2 = 4, 2
val1.__pow__(val2)       # 16
val2.__rpow__(val1)      # 16
# in-place power operation isn't supported by immutable classes like int, float, complex:
# val1.__ipow__(val2)
```

Exposiciónción utilizando el módulo matemático: math.pow ()

El módulo `math` contiene otra función `math.pow()` . La diferencia con el operador incorporado `pow()` -

function o `**` es que el resultado es siempre un `float` :

```
import math
math.pow(2, 2)    # 4.0
math.pow(-2., 2)  # 4.0
```

Lo que excluye los cálculos con entradas complejas:

```
math.pow(2, 2+0j)
```

TypeError: no se puede convertir complejo a flotar

y cálculos que conducirían a resultados complejos:

```
math.pow(-2, 0.5)
```

ValueError: error de dominio matemático

Función exponencial: math.exp () y cmath.exp ()

Tanto el módulo `math` como el módulo `cmath` contienen el número de Euler: e, y usarlo con la función `pow()` incorporada o el operador `**` funciona principalmente como `math.exp()` :

```
import math

math.e ** 2  # 7.3890560989306495
math.exp(2)  # 7.38905609893065

import cmath
cmath.e ** 2 # 7.3890560989306495
cmath.exp(2) # (7.38905609893065+0j)
```

Sin embargo, el resultado es diferente y usar la función exponencial directamente es más confiable que la exponenciación `math.e` con base `math.e` :

```
print(math.e ** 10)       # 22026.465794806703
print(math.exp(10))       # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
#     difference starts here ---------------^
```

Función exponencial menos 1: math.expm1 ()

El módulo `math` contiene la función `expm1()` que puede calcular la expresión `math.e ** x - 1` para `x` muy pequeña con mayor precisión que `math.exp(x)` o `cmath.exp(x)` permitiría:

```
import math

print(math.e ** 1e-3 - 1)  # 0.0010005001667083846
print(math.exp(1e-3) - 1)  # 0.0010005001667083846
print(math.expm1(1e-3))    # 0.0010005001667083417
#                            ------------------^
```

Para x muy pequeño la diferencia se hace más grande:

```
print(math.e ** 1e-15 - 1) # 1.1102230246251565e-15
print(math.exp(1e-15) - 1) # 1.1102230246251565e-15
print(math.expm1(1e-15))   # 1.0000000000000007e-15
#                            ^-------------------
```

La mejora es significativa en la computación científica. Por ejemplo, la ley de Planck contiene una función exponencial menos 1:

```
def planks_law(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * math.expm1(h * c / (lambda_ * k * T)))

def planks_law_naive(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * (math.e ** (h * c / (lambda_ * k * T)) - 1))

planks_law(100, 5000)        # 4.139080074896474e-19
planks_law_naive(100, 5000)  # 4.139080073488451e-19
#                                      ^----------

planks_law(1000, 5000)       # 4.139080128493406e-23
planks_law_naive(1000, 5000) # 4.139080233183142e-23
#                                    ^------------
```

Métodos mágicos y exponenciales: incorporados, matemáticos y matemáticos.

Suponiendo que tienes una clase que almacena valores puramente enteros:

```
class Integer(object):
    def __init__(self, value):
        self.value = int(value) # Cast to an integer

    def __repr__(self):
        return '{cls}({val})'.format(cls=self.__class__.__name__,
                                     val=self.value)

    def __pow__(self, other, modulo=None):
        if modulo is None:
            print('Using __pow__')
            return self.__class__(self.value ** other)
        else:
            print('Using __pow__ with modulo')
            return self.__class__(pow(self.value, other, modulo))

    def __float__(self):
        print('Using __float__')
        return float(self.value)

    def __complex__(self):
        print('Using __complex__')
        return complex(self.value, 0)
```

Usando la función `pow` incorporada o el operador `**` siempre llama a `__pow__` :

```
Integer(2) ** 2                 # Integer(4)
# Prints: Using __pow__
Integer(2) ** 2.5               # Integer(5)
# Prints: Using __pow__
pow(Integer(2), 0.5)            # Integer(1)
# Prints: Using __pow__
operator.pow(Integer(2), 3)     # Integer(8)
# Prints: Using __pow__
operator.__pow__(Integer(3), 3) # Integer(27)
# Prints: Using __pow__
```

El segundo argumento del método `__pow__()` solo se puede suministrar usando la función builtinpow `pow()` o llamando directamente al método:

```
pow(Integer(2), 3, 4)           # Integer(0)
# Prints: Using __pow__ with modulo
Integer(2).__pow__(3, 4)        # Integer(0)
# Prints: Using __pow__ with modulo
```

Mientras que las funciones `math` siempre lo convierten en un `float` y usan el cálculo de flotación:

```
import math

math.pow(Integer(2), 0.5) # 1.4142135623730951
# Prints: Using __float__
```

`cmath` funciones intentan convertirlo en `complex` pero también pueden retroceder a `float` si no hay una conversión explícita a `complex` :

```
import cmath

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __complex__

del Integer.__complex__   # Deleting __complex__ method - instances cannot be cast to complex

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __float__
```

Ni `math` ni `cmath` funcionarán si también `__float__()` el `__float__()` :

```
del Integer.__float__  # Deleting __complex__ method

math.sqrt(Integer(2))  # also cmath.exp(Integer(2))
```

TypeError: se requiere un flotador

Exponenciación modular: pow () con 3 argumentos.

Al suministrar `pow()` con 3 argumentos `pow(a, b, c)` evalúa la exponenciación modular $a^b \bmod c$:

```
pow(3, 4, 17)   # 13
```

```
# equivalent unoptimized expression:
3 ** 4 % 17     # 13

# steps:
3 ** 4          # 81
81 % 17         # 13
```

Para los tipos incorporados, el uso de exponenciación modular solo es posible si:

- El primer argumento es un `int`
- El segundo argumento es un `int >= 0`
- El tercer argumento es un `int != 0`

Estas restricciones también están presentes en Python 3.x

Por ejemplo, uno puede usar la forma de 3 argumentos de `pow` para definir una función inversa modular :

```
def modular_inverse(x, p):
    """Find a such as  a·x ≡ 1 (mod p), assuming p is prime."""
    return pow(x, p-2, p)

[modular_inverse(x, 13) for x in range(1,13)]
# Out: [1, 7, 9, 10, 8, 11, 2, 5, 3, 4, 6, 12]
```

Raíces: raíz nth con exponentes fraccionarios

Si bien la función `math.sqrt` se proporciona para el caso específico de raíces cuadradas, a menudo es conveniente usar el operador de exponenciación (`**`) con exponentes fraccionarios para realizar operaciones de raíz nth, como las raíces cúbicas.

La inversa de una exponenciación es la exponenciación por el recíproco del exponente. Entonces, si puedes calcular un número colocándolo en el exponente de 3, puedes encontrar la raíz cúbica de un número poniéndolo en el exponente de 1/3.

```
>>> x = 3
>>> y = x ** 3
>>> y
27
>>> z = y ** (1.0 / 3)
>>> z
3.0
>>> z == x
True
```

Cálculo de grandes raíces enteras

A pesar de que Python admite de forma nativa grandes enteros, tomar la raíz n de números muy grandes puede fallar en Python.

```
x = 2 ** 100
cube = x ** 3
```

```
root = cube ** (1.0 / 3)
```

OverflowError: largo int demasiado grande para convertirlo en flotante

Cuando trate con enteros tan grandes, necesitará usar una función personalizada para calcular la enésima raíz de un número.

```
def nth_root(x, n):
    # Start with some reasonable bounds around the nth root.
    upper_bound = 1
    while upper_bound ** n <= x:
        upper_bound *= 2
    lower_bound = upper_bound // 2
    # Keep searching for a better result as long as the bounds make sense.
    while lower_bound < upper_bound:
        mid = (lower_bound + upper_bound) // 2
        mid_nth = mid ** n
        if lower_bound < mid and mid_nth < x:
            lower_bound = mid
        elif upper_bound > mid and mid_nth > x:
            upper_bound = mid
        else:
            # Found perfect nth root.
            return mid
    return mid + 1

x = 2 ** 100
cube = x ** 3
root = nth_root(cube, 3)
x == root
# True
```

Lea Exposiciónción en línea: https://riptutorial.com/es/python/topic/347/exposicioncion

Capítulo 81: Expresiones Regulares (Regex)

Introducción

Python hace que las expresiones regulares estén disponibles a través del módulo `re` .

Las expresiones regulares son combinaciones de caracteres que se interpretan como reglas para hacer coincidir subcadenas. Por ejemplo, la expresión `'amount\D+\d+'` coincidirá con cualquier cadena compuesta por la `amount` palabra más un número integral, separados por uno o más no dígitos, como: `amount=100` , `amount is 3` , `amount is equal to: 33` , etc.

Sintaxis

- **Expresiones regulares directas**
- re.match (patrón, cadena, marca = 0) # Fuera: coincide con el patrón al principio de la cadena o Ninguno
- re.search (patrón, cadena, marca = 0) # Fuera: coincide con el patrón dentro de la cadena o Ninguno
- re.findall (pattern, string, flag = 0) # Out: lista de todas las coincidencias de pattern en string o []
- re.finditer (patrón, cadena, flag = 0) # Out: igual que re.findall, pero devuelve el objeto iterador
- re.sub (patrón, reemplazo, cadena, marca = 0) # Out: cadena con reemplazo (cadena o función) en lugar de patrón
- **Expresiones regulares precompiladas**
- precompiled_pattern = re.compile (patrón, flag = 0)
- precompiled_pattern.match (string) # Out: coincide al principio de string o Ninguno
- precompiled_pattern.search (string) # Out: coincide en cualquier lugar con string o None
- precompiled_pattern.findall (string) # Out: lista de todas las subcadenas coincidentes
- precompiled_pattern.sub (cadena / patrón / función, cadena) # Fuera: cadena reemplazada

Examples

Coincidiendo con el comienzo de una cadena

El primer argumento de `re.match()` es la expresión regular, el segundo es la cadena que debe

coincidir:

```
import re

pattern = r"123"
string = "123zzb"

re.match(pattern, string)
# Out: <_sre.SRE_Match object; span=(0, 3), match='123'>

match = re.match(pattern, string)

match.group()
# Out: '123'
```

Puede observar que la variable de patrón es una cadena con el prefijo `r` , que indica que la cadena es un *literal de cadena sin formato* .

Una cadena prima literal tiene una sintaxis ligeramente diferente de una cadena literal, es decir, una barra invertida `\` en un medio literal de cadena en bruto "sólo una barra invertida" y no hay necesidad de duplicar los juegos de escapar "secuencias de escape", tales como saltos de línea (`\n`) , tabulaciones (`\t`), espacios en blanco (`\`), feeds de formularios (`\r`), etc. En los literales de cadena normales, cada barra invertida debe duplicarse para evitar que se tome como el inicio de una secuencia de escape.

Por lo tanto, `r"\n"` es una cadena de 2 caracteres: `\` y `n` . Los patrones Regex también usan barras invertidas, por ejemplo, `\d` refiere a cualquier carácter de dígito. Podemos evitar tener que doble escape de nuestras cadenas (`"\\d"`) mediante el uso de cadenas sin formato (`r"\d"`).

Por ejemplo:

```
string = "\\t123zzb" # here the backslash is escaped, so there's no tab, just '\' and 't'
pattern = "\\t123"   # this will match \t (escaping the backslash) followed by 123
re.match(pattern, string).group()   # no match
re.match(pattern, "\t123zzb").group()  # matches '\t123'

pattern = r"\\t123"
re.match(pattern, string).group()   # matches '\\t123'
```

La coincidencia se realiza desde el principio de la cadena solamente. Si quieres hacer coincidir en cualquier lugar, usa `re.search` en `re.search` lugar:

```
match = re.match(r"(123)", "a123zzb")

match is None
# Out: True

match = re.search(r"(123)", "a123zzb")

match.group()
# Out: '123'
```

buscando

```
pattern = r"(your base)"
sentence = "All your base are belong to us."

match = re.search(pattern, sentence)
match.group(1)
# Out: 'your base'

match = re.search(r"(belong.*)", sentence)
match.group(1)
# Out: 'belong to us.'
```

La búsqueda se realiza en cualquier parte de la cadena a diferencia de `re.match` . También puedes usar `re.findall` .

También puede buscar al principio de la cadena (use `^`),

```
match = re.search(r"^123", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^123", "a123zzb")
match is None
# Out: True
```

al final de la cadena (use `$`),

```
match = re.search(r"123$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"123$", "123zzb")
match is None
# Out: True
```

o ambos (use ambos `^` y `$`):

```
match = re.search(r"^123$", "123")
match.group(0)
# Out: '123'
```

Agrupamiento

La agrupación se realiza entre paréntesis. El `group()` llamada `group()` devuelve una cadena formada por los subgrupos paréntesis coincidentes.

```
match.group() # Group without argument returns the entire match found
# Out: '123'
match.group(0) # Specifying 0 gives the same result as specifying no argument
# Out: '123'
```

También se pueden proporcionar argumentos a `group()` para obtener un subgrupo en particular.

De la documentación :

> Si hay un solo argumento, el resultado es una sola cadena; Si hay varios argumentos, el resultado es una tupla con un elemento por argumento.

Al llamar a `groups()` por otro lado, devuelve una lista de tuplas que contienen los subgrupos.

```
sentence = "This is a phone number 672-123-456-9910"
pattern = r".*(phone).*?([\d-]+)"

match = re.match(pattern, sentence)

match.groups()   # The entire match as a list of tuples of the paranthesized subgroups
# Out: ('phone', '672-123-456-9910')

m.group()        # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(0)       # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(1)       # The first parenthesized subgroup.
# Out: 'phone'

m.group(2)       # The second parenthesized subgroup.
# Out: '672-123-456-9910'

m.group(1, 2)    # Multiple arguments give us a tuple.
# Out: ('phone', '672-123-456-9910')
```

Grupos nombrados

```
match = re.search(r'My name is (?P<name>[A-Za-z ]+)', 'My name is John Smith')
match.group('name')
# Out: 'John Smith'

match.group(1)
# Out: 'John Smith'
```

Crea un grupo de captura al que se puede hacer referencia por nombre y por índice.

Grupos no capturadores

Usar `(?:)` crea un grupo, pero el grupo no se captura. Esto significa que puede usarlo como grupo, pero no contaminará su "espacio grupal".

```
re.match(r'(\d+)(\+(\d+))?', '11+22').groups()
# Out: ('11', '+22', '22')
```

```
re.match(r'(\d+)(?:\+(\d+))?', '11+22').groups()
# Out: ('11', '22')
```

Este ejemplo coincide con `11+22` u `11` , pero no con `11+` . Esto se debe a que el signo `+` y el segundo término están agrupados. Por otro lado, el signo `+` no es capturado.

Escapar de personajes especiales

Los caracteres especiales (como los corchetes de la clase de caracteres `[` y `]` continuación) no se corresponden literalmente:

```
match = re.search(r'[b]', 'a[b]c')
match.group()
# Out: 'b'
```

Al escapar de los caracteres especiales, pueden emparejarse literalmente:

```
match = re.search(r'\[b\]', 'a[b]c')
match.group()
# Out: '[b]'
```

La función `re.escape()` se puede utilizar para hacer esto por usted:

```
re.escape('a[b]c')
# Out: 'a\\[b\\]c'
match = re.search(re.escape('a[b]c'), 'a[b]c')
match.group()
# Out: 'a[b]c'
```

La función `re.escape()` escapa a todos los caracteres especiales, por lo que es útil si está componiendo una expresión regular basada en la entrada del usuario:

```
username = 'A.C.'  # suppose this came from the user
re.findall(r'Hi {}!'.format(username), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!', 'Hi ABCD!']
re.findall(r'Hi {}!'.format(re.escape(username)), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!']
```

Reemplazo

Los reemplazos se pueden hacer en cadenas usando `re.sub` .

Reemplazo de cuerdas

```
re.sub(r"t[0-9][0-9]", "foo", "my name t13 is t44 what t99 ever t44")
# Out: 'my name foo is foo what foo ever foo'
```

Usando referencias de grupo

Los reemplazos con un pequeño número de grupos se pueden hacer de la siguiente manera:

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Sin embargo, si crea una ID de grupo como '10', esto no funciona : `\10` se lee como 'ID número 1 seguido de 0'. Por lo tanto, debe ser más específico y usar la notación `\g<i>` :

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Usando una función de reemplazo

```
items = ["zero", "one", "two"]
re.sub(r"a\[([0-3])\]", lambda match: items[int(match.group(1))], "Items: a[0], a[1],
something, a[2]")
# Out: 'Items: zero, one, something, two'
```

Encontrar todos los partidos no superpuestos

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Out: ['12', '945', '444', '558', '889']
```

Tenga en cuenta que la `r` antes de `"[0-9]{2,3}"` le dice a python que interprete la cadena como está; como una cadena "sin procesar".

También puede usar `re.finditer()` que funciona de la misma manera que `re.findall()` pero devuelve un iterador con objetos `SRE_Match` lugar de una lista de cadenas:

```
results = re.finditer(r"([0-9]{2,3})", "some 1 text 12 is 945 here 4445588899")
print(results)
# Out: <callable-iterator object at 0x105245890>
for result in results:
     print(result.group(0))
''' Out:
12
945
444
558
889
'''
```

Patrones precompilados

```
import re
```

```
precompiled_pattern = re.compile(r"(\d+)")
matches = precompiled_pattern.search("The answer is 41!")
matches.group(1)
# Out: 41

matches = precompiled_pattern.search("Or was it 42?")
matches.group(1)
# Out: 42
```

La compilación de un patrón permite su reutilización posterior en un programa. Sin embargo, tenga en cuenta que Python almacena en caché las expresiones utilizadas recientemente (docs , SO answer), por lo que *"los programas que usan solo unas pocas expresiones regulares a la vez no tienen que preocuparse de compilar expresiones regulares"* .

```
import re

precompiled_pattern = re.compile(r"(.*\d+)")
matches = precompiled_pattern.match("The answer is 41!")
print(matches.group(1))
# Out: The answer is 41

matches = precompiled_pattern.match("Or was it 42?")
print(matches.group(1))
# Out: Or was it 42
```

Se puede utilizar con re.match ().

Comprobación de caracteres permitidos

Si desea comprobar que una cadena contiene solo un determinado conjunto de caracteres, en este caso az, AZ y 0-9, puede hacerlo así:

```
import re

def is_allowed(string):
    characherRegex = re.compile(r'[^a-zA-Z0-9.]')
    string = characherRegex.search(string)
    return not bool(string)

print (is_allowed("abyzABYZ0099"))
# Out: 'True'

print (is_allowed("#*@#$%^"))
# Out: 'False'
```

También puede adaptar la línea de expresión de `[^a-zA-Z0-9.]` `[^a-z0-9.]` , Por ejemplo, para no permitir letras en mayúsculas.

Crédito parcial: http://stackoverflow.com/a/1325265/2697955

Dividir una cadena usando expresiones regulares

También puedes usar expresiones regulares para dividir una cadena. Por ejemplo,

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print( data )
# Output: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

Banderas

Para algunos casos especiales, necesitamos cambiar el comportamiento de la Expresión regular, esto se hace usando indicadores. Los indicadores se pueden establecer de dos maneras, a través de la palabra clave de `flags` o directamente en la expresión.

Bandera de palabras clave

Debajo de un ejemplo para `re.search` pero funciona para la mayoría de las funciones en el módulo `re` .

```
m = re.search("b", "ABC")
m is None
# Out: True

m = re.search("b", "ABC", flags=re.IGNORECASE)
m.group()
# Out: 'B'

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE)
m is None
# Out: True

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE|re.DOTALL)
m.group()
# Out: 'A\nB'
```

Banderas comunes

Bandera	Breve descripción
`re.IGNORECASE` , `re.I`	Hace que el patrón ignore el caso.
`re.DOTALL` , `re.S`	Hace `.` combina todo, incluyendo nuevas líneas
`re.MULTILINE` , `re.M`	Hace `^` coincida con el comienzo de una línea y `$` el final de una línea
`re.DEBUG`	Activa la información de depuración

Para la lista completa de todas las banderas disponibles verifique los documentos

Banderas en linea

De la documentación :

```
(?iLmsux)
```

(Una o más letras del conjunto 'i', 'L', 'm', 's', 'u', 'x'.)

El grupo coincide con la cadena vacía; las letras establecen los indicadores correspondientes: re.I (ignorar mayúsculas y minúsculas), re.L (dependiente del entorno local), re.M (multilínea), re.S (punto corresponde a todos), re.U (dependiente de Unicode) y re.X (verbose), para toda la expresión regular. Esto es útil si desea incluir las banderas como parte de la expresión regular, en lugar de pasar un argumento de bandera a la función re.compile ().

Tenga en cuenta que el indicador (? X) cambia la forma en que se analiza la expresión. Debe usarse primero en la cadena de expresión o después de uno o más caracteres de espacio en blanco. Si hay caracteres que no son espacios en blanco antes de la bandera, los resultados no están definidos.

Iterando sobre los partidos usando `re.finditer`

Puede usar `re.finditer` para iterar sobre todas las coincidencias en una cadena. Esto le proporciona (en comparación con `re.findall` información adicional, como información sobre la ubicación de coincidencia en la cadena (índices):

```
import re
text = 'You can try to find an ant in this string'
pattern = 'an?\w' # find 'an' either with or without a following word character

for match in re.finditer(pattern, text):
    # Start index of match (integer)
    sStart = match.start()

    # Final index of match (integer)
    sEnd = match.end()

    # Complete match (string)
    sGroup = match.group()

    # Print match
    print('Match "{}" found at: [{},{}]'.format(sGroup, sStart,sEnd))
```

Resultado:

```
Match "an" found at: [5,7]
Match "an" found at: [20,22]
Match "ant" found at: [23,26]
```

Unir una expresión solo en lugares específicos

A menudo, usted quiere hacer coincidir una expresión solo en lugares *específicos* (dejándolos intactos en otros, es decir). Considera la siguiente oración:

```
An apple a day keeps the doctor away (I eat an apple everyday).
```

Aquí, la "manzana" aparece dos veces, lo que se puede resolver con los llamados *verbos de*

control de seguimiento inverso que son compatibles con el módulo de `regex` más nuevo. La idea es:

```
forget_this | or this | and this as well | (but keep this)
```

Con nuestro ejemplo de manzana, esto sería:

```
import regex as re
string = "An apple a day keeps the doctor away (I eat an apple everyday)."
rx = re.compile(r'''
    \([^()]*\) (*SKIP)(*FAIL)  # match anything in parentheses and "throw it away"
    |                          # or
    apple                      # match an apple
    ''', re.VERBOSE)
apples = rx.findall(string)
print(apples)
# only one
```

Esto coincide con "manzana" solo cuando se puede encontrar fuera de los paréntesis.

Así es como funciona:

- Mientras mira de **izquierda a derecha** , el motor de expresiones regulares consume todo a la izquierda, `(*SKIP)` actúa como una "afirmación de siempre verdadera". Después, falla correctamente en `(*FAIL)` y retrocede.
- Ahora llega al punto de `(*SKIP)` **de derecha a izquierda** (también conocido como retroceso) en el que está prohibido ir más hacia la izquierda. En su lugar, se le dice al motor que tire cualquier cosa a la izquierda y salte al punto donde se invocó `(*SKIP)` .

Lea Expresiones Regulares (Regex) en línea:
https://riptutorial.com/es/python/topic/632/expresiones-regulares--regex-

Capítulo 82: Extensiones de escritura

Examples

Hola mundo con extensión C

El siguiente archivo fuente de C (que llamaremos `hello.c` para propósitos de demostración) produce un módulo de extensión llamado `hello` que contiene una función simple `greet()` :

```
#include <Python.h>
#include <stdio.h>

#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif

static PyObject *hello_greet(PyObject *self, PyObject *args)
{
    const char *input;
    if (!PyArg_ParseTuple(args, "s", &input)) {
        return NULL;
    }
    printf("%s", input);
    Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] = {
    { "greet", hello_greet, METH_VARARGS, "Greet the user" },
    { NULL, NULL, 0, NULL }
};

#ifdef IS_PY3K
static struct PyModuleDef hellomodule = {
    PyModuleDef_HEAD_INIT, "hello", NULL, -1, HelloMethods
};

PyMODINIT_FUNC PyInit_hello(void)
{
    return PyModule_Create(&hellomodule);
}
#else
PyMODINIT_FUNC inithello(void)
{
    (void) Py_InitModule("hello", HelloMethods);
}
#endif
```

Para compilar el archivo con el compilador `gcc` , ejecute el siguiente comando en su terminal favorito:

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

Para ejecutar la función `greet()` que escribimos anteriormente, cree un archivo en el mismo directorio y `hello.py`

```
import hello          # imports the compiled library
hello.greet("Hello!") # runs the greet() function with "Hello!" as an argument
```

Pasando un archivo abierto a C Extensions

Pase un objeto de archivo abierto de Python a código de extensión C.

Puede convertir el archivo a un descriptor de archivo entero usando la función `PyObject_AsFileDescriptor` :

```
PyObject *fobj;
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0){
    return NULL;
}
```

Para convertir un descriptor de archivo entero de nuevo en un objeto de Python, use `PyFile_FromFd` .

```
int fd; /* Existing file descriptor */
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

Extensión C usando c ++ y Boost

Este es un ejemplo básico de una *extensión* C que usa C ++ y Boost .

Código C ++

Código C ++ puesto en hello.cpp:

```
#include <boost/python/module.hpp>
#include <boost/python/list.hpp>
#include <boost/python/class.hpp>
#include <boost/python/def.hpp>

// Return a hello world string.
std::string get_hello_function()
{
   return "Hello world!";
}

// hello class that can return a list of count hello world strings.
class hello_class
{
public:

   // Taking the greeting message in the constructor.
   hello_class(std::string message) : _message(message) {}

   // Returns the message count times in a python list.
   boost::python::list as_list(int count)
   {
```

```
      boost::python::list res;
      for (int i = 0; i < count; ++i) {
         res.append(_message);
      }
      return res;
   }

private:
   std::string _message;
};


// Defining a python module naming it to "hello".
BOOST_PYTHON_MODULE(hello)
{
   // Here you declare what functions and classes that should be exposed on the module.

   // The get_hello_function exposed to python as a function.
   boost::python::def("get_hello", get_hello_function);

   // The hello_class exposed to python as a class.
   boost::python::class_<hello_class>("Hello", boost::python::init<std::string>())
      .def("as_list", &hello_class::as_list)
      ;
}
```

Para compilar esto en un módulo de python, necesitarás los encabezados de python y las bibliotecas boost. Este ejemplo se hizo en Ubuntu 12.04 usando python 3.4 y gcc. Boost es compatible con muchas plataformas. En el caso de Ubuntu, los paquetes necesarios se instalaron usando:

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

Compilando el archivo fuente en un archivo .so que luego se puede importar como un módulo siempre que esté en la ruta de acceso de python:

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -
lboost_system -l:libpython3.4m.so
```

El código de python en el archivo example.py:

```
import hello

print(hello.get_hello())

h = hello.Hello("World hello!")
print(h.as_list(3))
```

Entonces `python3 example.py` dará el siguiente resultado:

```
Hello world!
['World hello!', 'World hello!', 'World hello!']
```

Lea Extensiones de escritura en línea: https://riptutorial.com/es/python/topic/557/extensiones-de-

escritura

Capítulo 83: Fecha y hora

Observaciones

Python proporciona métodos integrados y bibliotecas externas para crear, modificar, analizar y manipular fechas y horas.

Examples

Análisis de una cadena en un objeto de fecha y hora compatible con la zona horaria

Python 3.2+ admite el formato `%z` al analizar una cadena en un objeto de `datetime` y `datetime` .

> `+HHMM` UTC en la forma `+HHMM` o `-HHMM` (cadena vacía si el objeto es ingenuo).

Python 3.x 3.2

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

Para otras versiones de Python, puede usar una biblioteca externa como `dateutil` , que hace que el análisis de una cadena con zona horaria en un objeto de `datetime` y `datetime` sea rápido.

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

La variable `dt` ahora es un objeto de `datetime` y `datetime` con el siguiente valor:

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

Aritmética de fecha simple

Las fechas no existen de forma aislada. Es común que necesite encontrar la cantidad de tiempo entre las fechas o determinar cuál será la fecha de mañana. Esto se puede lograr usando objetos `timedelta`

```
import datetime

today = datetime.date.today()
print('Today:', today)

yesterday = today - datetime.timedelta(days=1)
print('Yesterday:', yesterday)

tomorrow = today + datetime.timedelta(days=1)
print('Tomorrow:', tomorrow)
```

```
print('Time between tomorrow and yesterday:', tomorrow - yesterday)
```

Esto producirá resultados similares a:

```
Today: 2016-04-15
Yesterday: 2016-04-14
Tomorrow: 2016-04-16
Difference between tomorrow and yesterday: 2 days, 0:00:00
```

Uso básico de objetos datetime

El módulo datetime contiene tres tipos principales de objetos: fecha, hora y fecha y hora.

```
import datetime

# Date object
today = datetime.date.today()
new_year = datetime.date(2017, 01, 01) #datetime.date(2017, 1, 1)

# Time object
noon = datetime.time(12, 0, 0) #datetime.time(12, 0)

# Current datetime
now = datetime.datetime.now()

# Datetime object
millenium_turn = datetime.datetime(2000, 1, 1, 0, 0, 0) #datetime.datetime(2000, 1, 1, 0, 0)
```

Las operaciones aritméticas para estos objetos solo se admiten dentro del mismo tipo de datos y realizar una aritmética simple con instancias de diferentes tipos resultará en un error de tipo.

```
# subtraction of noon from today
noon-today
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.date'
However, it is straightforward to convert between types.

# Do this instead
print('Time since the millenium at midnight: ',
      datetime.datetime(today.year, today.month, today.day) - millenium_turn)

# Or this
print('Time since the millenium at noon: ',
      datetime.datetime.combine(today, noon) - millenium_turn)
```

Iterar sobre fechas

A veces desea iterar en un rango de fechas desde una fecha de inicio hasta una fecha de finalización. Puedes hacerlo usando la biblioteca `datetime` y el objeto `timedelta` :

```
import datetime
```

```
# The size of each step in days
day_delta = datetime.timedelta(days=1)

start_date = datetime.date.today()
end_date = start_date + 7*day_delta

for i in range((end_date - start_date).days):
    print(start_date + i*day_delta)
```

Lo que produce:

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

Analizar una cadena con un nombre de zona horaria corto en un objeto de fecha y hora con fecha horaria

Al usar la biblioteca `dateutil` como en el ejemplo anterior en el análisis de las marcas de tiempo `dateutil` en la zona horaria , también es posible analizar las marcas de tiempo con un nombre de zona horaria "corto" especificado.

Para fechas formateadas con nombres de zonas horarias cortas o abreviaturas, que generalmente son ambiguas (p. Ej., CST, que podrían ser la Hora estándar central, la Hora estándar de China, la Hora estándar de Cuba, etc.) o puede encontrar más información aquí o no necesariamente están disponibles en una base de datos estándar , es necesario especificar una asignación entre la abreviatura de zona horaria y el objeto `tzinfo` .

```
from dateutil import tz
from dateutil.parser import parse

ET = tz.gettz('US/Eastern')
CT = tz.gettz('US/Central')
MT = tz.gettz('US/Mountain')
PT = tz.gettz('US/Pacific')

us_tzinfos = {'CST': CT, 'CDT': CT,
              'EST': ET, 'EDT': ET,
              'MST': MT, 'MDT': MT,
              'PST': PT, 'PDT': PT}

dt_est = parse('2014-01-02 04:00:00 EST', tzinfos=us_tzinfos)
dt_pst = parse('2016-03-11 16:00:00 PST', tzinfos=us_tzinfos)
```

Después de ejecutar esto:

```
dt_est
# datetime.datetime(2014, 1, 2, 4, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Eastern'))
dt_pst
```

```
# datetime.datetime(2016, 3, 11, 16, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Pacific'))
```

Vale la pena señalar que si utiliza una zona horaria de `pytz` con este método, *no* se localizará correctamente:

```
from dateutil.parser import parse
import pytz

EST = pytz.timezone('America/New_York')
dt = parse('2014-02-03 09:17:00 EST', tzinfos={'EST': EST})
```

Esto simplemente adjunta la zona horaria de `pytz` a la fecha y hora:

```
dt.tzinfo # Will be in Local Mean Time!
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Si utiliza este método, probablemente debería volver a `localize` la parte ingenua de la fecha y hora después del análisis:

```
dt_fixed = dt.tzinfo.localize(dt.replace(tzinfo=None))
dt_fixed.tzinfo # Now it's EST.
# <DstTzInfo 'America/New_York' EST-1 day, 19:00:00 STD>)
```

Construyendo tiempos de datos conscientes de la zona horaria

Por defecto, todos los objetos de `datetime` y `datetime` son ingenuos. Para que sean conscientes de la zona horaria, debe adjuntar un objeto `tzinfo` , que proporciona el desplazamiento UTC y la abreviatura de la zona horaria en función de la fecha y la hora.

Zonas horarias de compensación

Para las zonas horarias que son un desplazamiento fijo de UTC, en Python 3.2+, el módulo `datetime` proporciona la clase de `timezone` , una implementación concreta de `tzinfo` , que toma un `timedelta` y un parámetro de nombre (opcional):

Python 3.x 3.2

```
from datetime import datetime, timedelta, timezone
JST = timezone(timedelta(hours=+9))

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00

print(dt.tzname())
# UTC+09:00

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=timezone(timedelta(hours=9), 'JST'))
print(dt.tzname)
# 'JST'
```

Para versiones de Python anteriores a 3.2, es necesario usar una biblioteca de terceros, como

`dateutil` . `dateutil` proporciona una clase equivalente, `tzoffset` , que (a partir de la versión 2.5.3) toma argumentos de la forma `dateutil.tz.tzoffset(tzname, offset)` , donde el `offset` se especifica en segundos:

Python 3.x 3.2
Python 2.x 2.7

```
from datetime import datetime, timedelta
from dateutil import tz

JST = tz.tzoffset('JST', 9 * 3600) # 3600 seconds per hour
dt = datetime(2015, 1, 1, 12, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00
print(dt.tzname)
# 'JST'
```

Zonas con horario de verano.

Para las zonas con horario de verano, las bibliotecas estándar de Python no proporcionan una clase estándar, por lo que es necesario utilizar una biblioteca de terceros. `pytz` y `dateutil` son bibliotecas populares que proporcionan clases de zona horaria.

Además de las zonas horarias estáticas, `dateutil` proporciona clases de zonas `dateutil` que utilizan el horario de verano (consulte la documentación del módulo `tz`). Puede usar el método `tz.gettz()` para obtener un objeto de zona horaria, que luego se puede pasar directamente al constructor de `datetime` :

```
from datetime import datetime
from dateutil import tz
local = tz.gettz() # Local time
PT = tz.gettz('US/Pacific') # Pacific time

dt_l = datetime(2015, 1, 1, 12, tzinfo=local) # I am in EST
dt_pst = datetime(2015, 1, 1, 12, tzinfo=PT)
dt_pdt = datetime(2015, 7, 1, 12, tzinfo=PT) # DST is handled automatically
print(dt_l)
# 2015-01-01 12:00:00-05:00
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-07-01 12:00:00-07:00
```

PRECAUCIÓN : A partir de la versión 2.5.3, `dateutil` no maneja `dateutil` ambiguos correctamente, y siempre se establecerá de forma predeterminada en la fecha *posterior* . No hay forma de construir un objeto con una `dateutil` represente la zona horaria, por ejemplo, `2015-11-01 1:30 EDT-4` , ya que esto es *durante* una transición de horario de verano.

Todos los casos de borde se manejan correctamente cuando se usa `pytz` , pero las zonas horarias de `pytz` *no se* deben `pytz` directamente a las zonas horarias a través del constructor. En su lugar, se debe adjuntar una zona horaria de `pytz` usando el método de `localize` la zona horaria:

```
from datetime import datetime, timedelta
```

```
import pytz

PT = pytz.timezone('US/Pacific')
dt_pst = PT.localize(datetime(2015, 1, 1, 12))
dt_pdt = PT.localize(datetime(2015, 11, 1, 0, 30))
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-11-01 00:30:00-07:00
```

Tenga en cuenta que si realiza aritmética de fecha y hora en una `pytz` horaria de `pytz` -aware, debe realizar los cálculos en UTC (si desea un tiempo transcurrido absoluto), o debe llamar a `normalize()` en el resultado:

```
dt_new = dt_pdt + timedelta(hours=3) # This should be 2:30 AM PST
print(dt_new)
# 2015-11-01 03:30:00-07:00
dt_corrected = PT.normalize(dt_new)
print(dt_corrected)
# 2015-11-01 02:30:00-08:00
```

Análisis de fecha y hora difuso (extracción de fecha y hora de un texto)

Es posible extraer una fecha de un texto usando el analizador de `dateutil` en un modo "borroso", donde los componentes de la cadena que no se reconocen como parte de una fecha se ignoran.

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

`dt` ahora es un *objeto* `datetime` y vería `datetime.datetime(2047, 1, 1, 8, 21)` impreso.

Cambio entre zonas horarias

Para cambiar entre zonas horarias, necesita objetos de fecha y hora que tengan en cuenta la zona horaria.

```
from datetime import datetime
from dateutil import tz

utc = tz.tzutc()
local = tz.tzlocal()

utc_now = datetime.utcnow()
utc_now # Not timezone-aware.

utc_now = utc_now.replace(tzinfo=utc)
utc_now # Timezone-aware.

local_now = utc_now.astimezone(local)
local_now # Converted to local time.
```

Análisis de una marca de tiempo ISO 8601 arbitraria con bibliotecas mínimas

Python solo tiene soporte limitado para analizar las marcas de tiempo ISO 8601. Para `strptime` necesita saber exactamente en qué formato se encuentra. Como complicación, la clasificación de una `datetime` y `datetime` es una marca de tiempo ISO 8601, con espacio como separador y fracción de 6 dígitos:

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))
# '2016-07-22 09:25:59.555555'
```

pero si la fracción es 0, no se genera ninguna parte fraccionaria

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

Pero estas 2 formas necesitan un formato *diferente* para `strptime` . Además, `strptime' does not support at all parsing minute timezones that have a : in it, thus can be parsed, but the standard format` 2016-07-22 09: 25: 59 + 0300 `can be parsed, but the standard format` 2016-07-22 09:25:59 +03: 00` no puedo.

Hay una biblioteca de un solo archivo llamada `iso8601` que analiza correctamente las marcas de tiempo ISO 8601 y solo ellas.

Admite fracciones y zonas horarias, y el separador en `T` todo con una sola función:

```
import iso8601
iso8601.parse_date('2016-07-22 09:25:59')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22 09:25:59+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<FixedOffset '+03:00' ...>)
iso8601.parse_date('2016-07-22 09:25:59Z')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22T09:25:59.000111+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, 111, tzinfo=<FixedOffset '+03:00' ...>)
```

Si no se establece una zona horaria, `iso8601.parse_date` se establece de `iso8601.parse_date` predeterminada en UTC. La zona predeterminada se puede cambiar con el argumento de palabra clave `default_zone` . En particular, si esto es `None` lugar del predeterminado, entonces las marcas de tiempo que no tienen una zona horaria explícita se devuelven como tiempos de referencia ingenuos:

```
iso8601.parse_date('2016-07-22T09:25:59', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59)
iso8601.parse_date('2016-07-22T09:25:59Z', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
```

Convertir la marca de tiempo a datetime

El módulo `datetime` puede convertir una `timestamp` de `timestamp` POSIX en un objeto `datetime` ITC.

La época es el 1 de enero de 1970 a medianoche.

```
import time
from datetime import datetime
seconds_since_epoch=time.time()  #1469182681.709

utc_date=datetime.utcfromtimestamp(seconds_since_epoch) #datetime.datetime(2016, 7, 22, 10,
18, 1, 709000)
```

Restar meses de una fecha con precisión

Usando el módulo de `calendar`

```
import calendar
from datetime import date

def monthdelta(date, delta):
    m, y = (date.month+delta) % 12, date.year + ((date.month)+delta-1) // 12
    if not m: m = 12
    d = min(date.day, calendar.monthrange(y, m)[1])
    return date.replace(day=d,month=m, year=y)

next_month = monthdelta(date.today(), 1) #datetime.date(2016, 10, 23)
```

Usando el módulo `dateutils`

```
import datetime
import dateutil.relativedelta

d = datetime.datetime.strptime("2013-03-31", "%Y-%m-%d")
d2 = d - dateutil.relativedelta.relativedelta(months=1)  #datetime.datetime(2013, 2, 28, 0, 0)
```

Calcular las diferencias de tiempo

`timedelta` módulo `timedelta` es útil para calcular las diferencias entre los tiempos:

```
from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23)     # datetime.datetime(2016, 05, 23, 0, 0, 0)
```

Especificar el tiempo es opcional al crear un nuevo objeto de `datetime`

```
delta = now-then
```

`delta` es de tipo `timedelta`

```
print(delta.days)
# 60
print(delta.seconds)
# 40826
```

Para obtener n día después y n día antes de la fecha podríamos usar:

día después de la fecha:

```
def get_n_days_after_date(date_format="%d %B %Y", add_days=120):

    date_n_days_after = datetime.datetime.now() + timedelta(days=add_days)
    return date_n_days_after.strftime(date_format)
```

n día anterior a la fecha:

```
 def get_n_days_before_date(self, date_format="%d %B %Y", days_before=120):

        date_n_days_ago = datetime.datetime.now() - timedelta(days=days_before)
        return date_n_days_ago.strftime(date_format)
```

Obtener una marca de tiempo ISO 8601

Sin zona horaria, con microsegundos.

```
from datetime import datetime

datetime.now().isoformat()
# Out: '2016-07-31T23:08:20.886783'
```

Con zona horaria, con microsegundos.

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# Out: '2016-07-31T23:09:43.535074-07:00'
```

Con zona horaria, sin microsegundos.

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Out: '2016-07-31T23:10:30-07:00'
```

Consulte ISO 8601 para obtener más información sobre el formato ISO 8601.

Lea Fecha y hora en línea: https://riptutorial.com/es/python/topic/484/fecha-y-hora

Capítulo 84: Filtrar

Sintaxis

- filtro (función, iterable)
- itertools.ifilter (función, iterable)
- future_builtins.filter (función, iterable)
- itertools.ifilterfalse (función, iterable)
- itertools.filterfalse (función, iterable)

Parámetros

Parámetro	Detalles
función	*llamable* que determina la condición o `None` luego use la función de identidad para el filtrado (*solo de posición*)
iterable	iterable que será filtrado (*solo posicional*)

Observaciones

En la mayoría de los casos, una expresión de comprensión o generador es más legible, más potente y más eficiente que `filter()` o `ifilter()` .

Examples

Uso básico del filtro.

Para `filter` descartan elementos de una secuencia en función de algunos criterios:

```
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5
```

Python 2.x 2.0

```
filter(long_name, names)
# Out: ['Barney']

[name for name in names if len(name) > 5] # equivalent list comprehension
# Out: ['Barney']


from itertools import ifilter
ifilter(long_name, names)       # as generator (similar to python 3.x filter builtin)
```

```
# Out: <itertools.ifilter at 0x4197e10>
list(ifilter(long_name, names)) # equivalent to filter with lists
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x0000000003FD5D38>
```

Python 2.x 2.6

```
# Besides the options for older python 2.x versions there is a future_builtin function:
from future_builtins import filter
filter(long_name, names)       # identical to itertools.ifilter
# Out: <itertools.ifilter at 0x3eb0ba8>
```

Python 3.x 3.0

```
filter(long_name, names)        # returns a generator
# Out: <filter at 0x1fc6e443470>
list(filter(long_name, names))  # cast to list
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x000001C6F49BF4C0>
```

Filtro sin función

Si el parámetro de la función es `None` , se utilizará la función de identidad:

```
list(filter(None, [1, 0, 2, [], '', 'a']))  # discards 0, [] and ''
# Out: [1, 2, 'a']
```

Python 2.x 2.0.1

```
[i for i in [1, 0, 2, [], '', 'a'] if i] # equivalent list comprehension
```

Python 3.x 3.0.0

```
(i for i in [1, 0, 2, [], '', 'a'] if i) # equivalent generator expression
```

Filtrar como comprobación de cortocircuito.

`filter` (3.x pitón) y `ifilter` (Python 2.x) devuelve un generador, así que puede ser muy útil cuando se crea un ensayo de cortocircuito como `or` o `and` :

Python 2.x 2.0.1

```
 # not recommended in real use but keeps the example short:
from itertools import ifilter as filter
```

Python 2.x 2.6.1

```
 from future_builtins import filter
```

Para encontrar el primer elemento que es más pequeño que 100:

```
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filter(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
#        Check rectangular tire, 80$
# Out: ('rectangular tire', 80)
```

La `next` función proporciona el siguiente elemento (en este caso, primero) y es, por lo tanto, la razón por la que es un cortocircuito.

Función complementaria: filterfalse, ifilterfalse

Hay una función complementaria para el `filter` en el módulo `itertools` :

Python 2.x 2.0.1

```
 # not recommended in real use but keeps the example valid for python 2.x and python 3.x
from itertools import ifilterfalse as filterfalse
```

Python 3.x 3.0.0

```
from itertools import filterfalse
```

que funciona exactamente igual que el `filter` del *generador* pero mantiene solo los elementos que son `False` :

```
# Usage without function (None):
list(filterfalse(None, [1, 0, 2, [], '', 'a']))  # discards 1, 2, 'a'
# Out: [0, [], '']
```

```
# Usage with function
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5

list(filterfalse(long_name, names))
# Out: ['Fred', 'Wilma']
```

```
# Short-circuit useage with next:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filterfalse(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
# Out: ('Toyota', 1000)
```

```
# Using an equivalent generator:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
generator = (car for car in car_shop if not car[1] < 100)
next(generator)
```

Lea Filtrar en línea: https://riptutorial.com/es/python/topic/201/filtrar

Capítulo 85: Formato de cadena

Introducción

Cuando se almacenan y transforman datos para que los humanos los vean, el formato de cadena puede ser muy importante. Python ofrece una amplia variedad de métodos de formato de cadenas que se describen en este tema.

Sintaxis

- "{}". formato (42) ==> "42"
- "{0}". Formato (42) ==> "42"
- "{0: .2f}". Formato (42) ==> "42.00"
- "{0: .0f}". Formato (42.1234) ==> "42"
- "{answer}". formato (no_answer = 41, respuesta = 42) ==> "42"
- "{answer: .2f}". format (no_answer = 41, answer = 42) ==> "42.00"
- "{[clave]}". formato ({'clave': 'valor'}) ==> "valor"
- "{[1]}". Formato (['cero', 'uno', 'dos']) ==> "uno"
- "{answer} = {answer}". formato (respuesta = 42) ==> "42 = 42"
- " .join (['stack', 'overflow']) ==> "stack overflow"

Observaciones

- Debería visitar PyFormat.info para una introducción / explicación muy completa y suave de cómo funciona.

Examples

Conceptos básicos de formato de cadena

```
foo = 1
bar = 'bar'
baz = 3.14
```

Puedes usar `str.format` para formatear la salida. Los pares de corchetes se reemplazan con argumentos en el orden en que se pasan los argumentos:

```
print('{}, {} and {}'.format(foo, bar, baz))
# Out: "1, bar and 3.14"
```

Los índices también se pueden especificar dentro de los corchetes. Los números corresponden a los índices de los argumentos pasados a la función `str.format` (basado en 0).

```
print('{0}, {1}, {2}, and {1}'.format(foo, bar, baz))
```

```
# Out: "1, bar, 3.14, and bar"
print('{0}, {1}, {2}, and {3}'.format(foo, bar, baz))
# Out: index out of range error
```

Los argumentos con nombre también se pueden utilizar:

```
print("X value is: {x_val}. Y value is: {y_val}.".format(x_val=2, y_val=3))
# Out: "X value is: 2. Y value is: 3."
```

Se puede hacer referencia a los atributos de los objetos cuando se pasan a `str.format` :

```
class AssignValue(object):
    def __init__(self, value):
        self.value = value
my_value = AssignValue(6)
print('My value is: {0.value}'.format(my_value))  # "0" is optional
# Out: "My value is: 6"
```

Las claves del diccionario se pueden utilizar también:

```
my_dict = {'key': 6, 'other_key': 7}
print("My other key is: {0[other_key]}".format(my_dict))  # "0" is optional
# Out: "My other key is: 7"
```

Lo mismo se aplica a los índices de lista y tupla:

```
my_list = ['zero', 'one', 'two']
print("2nd element is: {0[2]}".format(my_list))  # "0" is optional
# Out: "2nd element is: two"
```

> Nota: Además de `str.format` , Python también proporciona el operador de módulo `%` también conocido como *operador de formateo* o *interpolación de cadenas* (ver PEP 3101), para el formato de cadenas. `str.format` es un sucesor de `%` y ofrece una mayor flexibilidad, por ejemplo, al facilitar la realización de múltiples sustituciones.

Además de los índices de argumentos, también puede incluir una *especificación de formato* dentro de los corchetes. Esta es una expresión que sigue reglas especiales y debe ser precedido por dos puntos (`:`). Consulte la documentación para obtener una descripción completa de la especificación de formato. Un ejemplo de especificación de formato es la directiva de alineación `:~^20` (`^` significa alineación central, ancho total 20, relleno con ~ carácter):

```
'{:~^20}'.format('centered')
# Out: '~~~~~~centered~~~~~~'
```

`format` permite un comportamiento no posible con `%` , por ejemplo, la repetición de argumentos:

```
t = (12, 45, 22222, 103, 6)
print '{0} {2} {1} {2} {3} {2} {4} {2}'.format(*t)
# Out: 12 22222 45 22222 103 22222 6 22222
```

Como el `format` es una función, se puede usar como un argumento en otras funciones:

```
number_list = [12,45,78]
print map('the number is {}'.format, number_list)
# Out: ['the number is 12', 'the number is 45', 'the number is 78']


from datetime import datetime,timedelta

once_upon_a_time = datetime(2010, 7, 1, 12, 0, 0)
delta = timedelta(days=13, hours=8,  minutes=20)

gen = (once_upon_a_time + x * delta for x in xrange(5))

print '\n'.join(map('{:%Y-%m-%d %H:%M:%S}'.format, gen))
#Out: 2010-07-01 12:00:00
#     2010-07-14 20:20:00
#     2010-07-28 04:40:00
#     2010-08-10 13:00:00
#     2010-08-23 21:20:00
```

Alineación y relleno.

Python 2.x 2.6

El método `format()` se puede usar para cambiar la alineación de la cadena. Tienes que hacerlo con una expresión de formato del formulario `:[fill_char][align_operator][width]` donde `align_operator` es uno de los siguientes:

- `<` obliga al campo a alinearse a la izquierda dentro del `width` .
- `>` obliga al campo a alinearse a la derecha dentro del `width` .
- `^` obliga al campo a centrarse dentro de la `width` .
- `=` obliga a que el relleno se coloque después del signo (solo tipos numéricos).

`fill_char` (si se omite, el valor predeterminado es el espacio en blanco) es el carácter utilizado para el relleno.

```
'{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

'{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'

'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'
```

Nota: puede obtener los mismos resultados utilizando las funciones de cadena `ljust()` , `rjust()` , `center()` , `zfill()` , sin embargo, estas funciones están en desuso desde la versión 2.5.

Formato literales (f-string)

Las cadenas de formato literal se introdujeron en PEP 498 (Python3.6 y versiones posteriores), lo que le permite anteponer `f` al comienzo de un literal de cadena para aplicar efectivamente el `.format` con todas las variables en el alcance actual.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

Esto también funciona con cadenas de formato más avanzadas, incluida la alineación y la notación de puntos.

```
>>> f'{foo:^7s}'
'  bar  '
```

Nota: La `f''` no denota un tipo particular como `b''` para `bytes` ou `u''` para `unicode` en python2. El formateo se aplica inmediatamente, lo que resulta en una agitación normal.

Las cadenas de formato también se pueden *anidar* :

```
>>> price = 478.23
>>> f"{f'${price:0.2f}':*>20s}"
'*************$478.23'
```

Las expresiones en una cadena-f se evalúan en orden de izquierda a derecha. Esto es detectable solo si las expresiones tienen efectos secundarios:

```
>>> def fn(l, incr):
...     result = l[0]
...     l[0] += incr
...     return result
...
>>> lst = [0]
>>> f'{fn(lst,2)} {fn(lst,3)}'
'0 2'
>>> f'{fn(lst,2)} {fn(lst,3)}'
'5 7'
>>> lst
[10]
```

Formato de cadena con fecha y hora

Cualquier clase puede configurar su propia sintaxis de formato de cadena a través del método `__format__` . Un tipo en la biblioteca estándar de Python que hace un uso práctico de esto es el tipo `datetime` , donde se pueden usar códigos de formato similares a `strftime` directamente dentro de `str.format` :

```
>>> from datetime import datetime
>>> 'North America: {dt:%m/%d/%Y}.  ISO: {dt:%Y-%m-%d}.'.format(dt=datetime.now())
'North America: 07/21/2016.  ISO: 2016-07-21.'
```

Se puede encontrar una lista completa de la lista de formateadores de fecha y hora en la

documentación oficial .

Formato utilizando Getitem y Getattr

Cualquier estructura de datos que admita `__getitem__` puede tener su estructura anidada formateada:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

A los atributos de los objetos se puede acceder usando `getattr()` :

```
class Person(object):
    first = 'Zaphod'
    last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

Formato flotante

```
>>> '{0:.0f}'.format(42.12345)
'42'

>>> '{0:.1f}'.format(42.12345)
'42.1'

>>> '{0:.3f}'.format(42.12345)
'42.123'

>>> '{0:.5f}'.format(42.12345)
'42.12345'

>>> '{0:.7f}'.format(42.12345)
'42.1234500'
```

Lo mismo vale para otra forma de referenciar:

```
>>> '{:.3f}'.format(42.12345)
'42.123'

>>> '{answer:.3f}'.format(answer=42.12345)
'42.123'
```

Los números de punto flotante también se pueden formatear en notación científica o como porcentajes:

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'

>>> '{0:.0%}'.format(42.12345)
'4212%'
```

También puede combinar las notaciones `{0}` y `{name}` . Esto es especialmente útil cuando desea redondear todas las variables a un número de decimales preespecificado *con 1 declaración* :

```
>>> s = 'Hello'
>>> a, b, c = 1.12345, 2.34567, 34.5678
>>> digits = 2

>>> '{0}! {1:.{n}f}, {2:.{n}f}, {3:.{n}f}'.format(s, a, b, c, n=digits)
'Hello! 1.12, 2.35, 34.57'
```

Formato de valores numéricos

El método `.format()` puede interpretar un número en diferentes formatos, tales como:

```
>>> '{:c}'.format(65)    # Unicode character
'A'

>>> '{:d}'.format(0x0a)  # base 10
'10'

>>> '{:n}'.format(0x0a)  # base 10 using current locale for separators
'10'
```

Formato de enteros a diferentes bases (hex, oct, binario)

```
>>> '{0:x}'.format(10) # base 16, lowercase – Hexadecimal
'a'

>>> '{0:X}'.format(10) # base 16, uppercase – Hexadecimal
'A'

>>> '{:o}'.format(10) # base 8 – Octal
'12'

>>> '{:b}'.format(10) # base 2 – Binary
'1010'

>>> '{0:#b}, {0:#o}, {0:#x}'.format(42) # With prefix
'0b101010, 0o52, 0x2a'

>>> '8 bit: {0:08b}; Three bytes: {0:06x}'.format(42) # Add zero padding
'8 bit: 00101010; Three bytes: 00002a'
```

Use el formato para convertir una tupla flotante RGB en una cadena hexadecimal de color:

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{:02X}{:02X}{:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'#FF6600'
```

Solo se pueden convertir enteros:

```
>>> '{:x}'.format(42.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ValueError: Unknown format code 'x' for object of type 'float'
```

Formato personalizado para una clase

Nota:

> Todo lo siguiente se aplica al método `str.format` , así como a la función de `format` . En el texto de abajo, los dos son intercambiables.

Para cada valor que se pasa a la función de `format` , Python busca un método `__format__` para ese argumento. Por lo tanto, su propia clase personalizada puede tener su propio método `__format__` para determinar cómo la función de `format` mostrará y formateará su clase y sus atributos.

Esto es diferente al método `__str__` , ya que en el método `__format__` puede tener en cuenta el idioma del formato, incluida la alineación, el ancho del campo, etc. e incluso (si lo desea) implementar sus propios especificadores de formato y sus propias extensiones de lenguaje de formato. 1

```
object.__format__(self, format_spec)
```

Por ejemplo :

```
# Example in Python 2 - but can be easily applied to Python 3

class Example(object):
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a,b,c

    def __format__(self, format_spec):
        """ Implement special semantics for the 's' format specifier """
        # Reject anything that isn't an s
        if format_spec[-1] != 's':
            raise ValueError('{} format specifier not understood for this object',
format_spec[:-1])

        # Output in this example will be (<a>,<b>,<c>)
        raw = "(" + ",".join([str(self.a), str(self.b), str(self.c)]) + ")"
        # Honor the format language by using the inbuilt string format
        # Since we know the original format_spec ends in an 's'
        # we can take advantage of the str.format method with a
        # string argument we constructed above
        return "{r:{f}}".format( r=raw, f=format_spec )

inst = Example(1,2,3)
print "{0:>20s}".format( inst )
# out :              (1,2,3)
# Note how the right align and field width of 20 has been honored.
```

Nota:

> Si su clase personalizada no tiene un método `__format__` personalizado y se `__format__` una instancia de la clase a la función de `format` , **Python2** siempre usará el valor de retorno del método `__str__` o el método `__repr__` para determinar qué imprimir (y si no

existe ninguno, entonces se utilizará la `repr` predeterminada), y deberá usar el especificador de formato `s` para formatear esto. Con **Python3** , para pasar su clase personalizada a la función de `format` , necesitará definir el método `__format__` en su clase personalizada.

Formateo anidado

Algunos formatos pueden tomar parámetros adicionales, como el ancho de la cadena con formato o la alineación:

```
>>> '{:.>10}'.format('foo')
'.......foo'
```

También se pueden proporcionar como parámetros para `format` anidando más `{}` dentro de `{}` :

```
>>> '{:.>{}}'.format('foo', 10)
'.......foo'
'{:{}{}{}}'.format('foo', '*', '^', 15)
'******foo******'
```

En el último ejemplo, la cadena de formato `'{:{}{}{}}'` se modifica a `'{:*^15}'` (es decir, "centro y pad con * a una longitud total de 15") antes de aplicarla a la cadena real `'foo'` para ser formateada de esa manera.

Esto puede ser útil en casos en que los parámetros no se conocen de antemano, por ejemplo al alinear datos tabulares:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
bbbbbbb
    ccc
```

Acolchado y cuerdas truncantes, combinadas.

Digamos que quieres imprimir variables en una columna de 3 caracteres.

Nota: doblando `{` y `}` se les escapa.

```
s = """

pad
{{:3}}             :{a:3}:

truncate
{{:.3}}            :{e:.3}:

combined
{{:>3.3}}          :{a:>3.3}:
{{:3.3}}           :{a:3.3}:
```

```
{{:3.3}}           :{c:3.3}:
{{:3.3}}           :{e:3.3}:
"""

print (s.format(a="1"*1, c="3"*3, e="5"*5))
```

Salida:

```
pad
{:3}             :1  :

truncate
{:.3}            :555:

combined
{:>3.3}          :  1:
{:3.3}           :1  :
{:3.3}           :333:
{:3.3}           :555:
```

Marcadores de posición nombrados

Las cadenas de formato pueden contener marcadores de posición con nombre que se interpolan usando argumentos de palabras clave para dar `format` .

Usando un diccionario (Python 2.x)

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

Usando un diccionario (Python 3.2+)

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

`str.format_map` permite usar diccionarios sin tener que descomprimirlos primero. También se usa la clase de `data` (que podría ser un tipo personalizado) en lugar de un `dict` recién llenado.

Sin un diccionario:

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
'Hodor Hodor!'
```

Lea Formato de cadena en línea: https://riptutorial.com/es/python/topic/1019/formato-de-cadena

Capítulo 86: Formato de fecha

Examples

Tiempo entre dos fechas

```
from datetime import datetime

a = datetime(2016,10,06,0,0,0)
b = datetime(2016,10,01,23,59,59)

a-b
# datetime.timedelta(4, 1)

(a-b).days
# 4
(a-b).total_seconds()
# 518399.0
```

Analizando la cadena al objeto datetime

Utiliza códigos de formato estándar C

```
from datetime import datetime
datetime_string = 'Oct 1 2016, 00:00:00'
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strptime(datetime_string, datetime_string_format)
# datetime.datetime(2016, 10, 1, 0, 0)
```

Salida de objetos de fecha y hora a cadena

Utiliza códigos de formato estándar C

```
from datetime import datetime
datetime_for_string = datetime(2016,10,1,0,0)
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strftime(datetime_for_string,datetime_string_format)
# Oct 01 2016, 00:00:00
```

Lea Formato de fecha en línea: https://riptutorial.com/es/python/topic/7284/formato-de-fecha

Capítulo 87: Función de mapa

Sintaxis

- mapa (función, iterable [, * additional_iterables])
- future_builtins.map (función, iterable [, * additional_iterables])
- itertools.imap (función, iterable [, * additional_iterables])

Parámetros

Parámetro	Detalles
función	función de mapeo (debe tomar tantos parámetros como haya iterables) (*solo de posición*)
iterable	la función se aplica a cada elemento de lo iterable (*solo posicional*)
* adicional_iterables	vea iterable, pero tantos como desee (*opcional* , *solo posicional*)

Observaciones

Todo lo que se puede hacer con el `map` también se puede hacer con `comprehensions` :

```
list(map(abs, [-1,-2,-3]))     # [1, 2, 3]
[abs(i) for i in [-1,-2,-3]]   # [1, 2, 3]
```

Aunque necesitarías `zip` si tienes múltiples iterables:

```
import operator
alist = [1,2,3]
list(map(operator.add, alist, alist))  # [2, 4, 6]
[i + j for i, j in zip(alist, alist)]  # [2, 4, 6]
```

Las comprensiones de listas son eficientes y pueden ser más rápidas que el `map` en muchos casos, así que pruebe los tiempos de ambos enfoques si la velocidad es importante para usted.

Examples

Uso básico de map, itertools.imap y future_builtins.map

La función de mapa es la más simple entre las incorporaciones de Python utilizadas para la programación funcional. `map()` aplica una función específica a cada elemento en un iterable:

```
names = ['Fred', 'Wilma', 'Barney']
```

Python 3.x 3.0

```
map(len, names)  # map in Python 3.x is a class; its instances are iterable
# Out: <map object at 0x00000198B32E2CF8>
```

Se incluye un `map` compatible con Python 3 en el módulo `future_builtins` :

Python 2.x 2.6

```
from future_builtins import map  # contains a Python 3.x compatible map()
map(len, names)                  # see below
# Out: <itertools.imap instance at 0x3eb0a20>
```

Alternativamente, en Python 2 se puede usar `imap` de `itertools` para obtener un generador

Python 2.x 2.3

```
map(len, names)   # map() returns a list
# Out: [4, 5, 6]

from itertools import imap
imap(len, names)  # itertools.imap() returns a generator
# Out: <itertools.imap at 0x405ea20>
```

El resultado se puede convertir explícitamente en una `list` para eliminar las diferencias entre Python 2 y 3:

```
list(map(len, names))
# Out: [4, 5, 6]
```

`map()` puede reemplazarse por una *lista* equivalente de *comprensión* o *expresión de generador* :

```
[len(item) for item in names] # equivalent to Python 2.x map()
# Out: [4, 5, 6]

(len(item) for item in names) # equivalent to Python 3.x map()
# Out: <generator object <genexpr> at 0x00000195888D5FC0>
```

Mapeando cada valor en una iterable

Por ejemplo, puedes tomar el valor absoluto de cada elemento:

```
list(map(abs, (1, -1, 2, -2, 3, -3))) # the call to `list` is unnecessary in 2.x
# Out: [1, 1, 2, 2, 3, 3]
```

Función anónima también soporte para mapear una lista:

```
map(lambda x:x*2, [1, 2, 3, 4, 5])
# Out: [2, 4, 6, 8, 10]
```

o convirtiendo valores decimales a porcentajes:

```
def to_percent(num):
    return num * 100

list(map(to_percent, [0.95, 0.75, 1.01, 0.1]))
# Out: [95.0, 75.0, 101.0, 10.0]
```

o convertir dólares a euros (dado un tipo de cambio):

```
from functools import partial
from operator import mul

rate = 0.9  # fictitious exchange rate, 1 dollar = 0.9 euros
dollars = {'under_my_bed': 1000,
           'jeans': 45,
           'bank': 5000}

sum(map(partial(mul, rate), dollars.values()))
# Out: 5440.5
```

`functools.partial` es una forma conveniente de corregir los parámetros de las funciones para que puedan usarse con `map` lugar de usar `lambda` o crear funciones personalizadas.

Mapeo de valores de diferentes iterables.

Por ejemplo, calculando el promedio de cada elemento `i` -ésimo de múltiples iterables:

```
def average(*args):
    return float(sum(args)) / len(args)  # cast to float - only mandatory for python 2.x

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117, 91, 102]
measurement3 = [104, 102, 95, 101]

list(map(average, measurement1, measurement2, measurement3))
# Out: [102.0, 110.0, 95.0, 100.0]
```

Hay diferentes requisitos si se pasa más de un iterable al `map` dependiendo de la versión de python:

- La función debe tener tantos parámetros como sean iterables:

  ```
  def median_of_three(a, b, c):
      return sorted((a, b, c))[1]

  list(map(median_of_three, measurement1, measurement2))
  ```

  TypeError: median_of_three () falta 1 argumento posicional requerido: 'c'

  ```
  list(map(median_of_three, measurement1, measurement2, measurement3, measurement3))
  ```

  TypeError: median_of_three () toma 3 argumentos posicionales pero se dieron 4

Python 2.x 2.0.1

- `map` : el mapeo se repite siempre y cuando un iterable aún no esté completamente consumido, pero no asuma `None` de los iterables totalmente consumidos:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
```

TypeError: tipo (s) de operando no compatibles para -: 'int' y 'NoneType'

- `itertools.imap` y `future_builtins.map` : la asignación se detiene tan pronto como se detiene una iterable:

```
import operator
from itertools import imap

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(imap(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(imap(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Python 3.x 3.0.0

- La asignación se detiene tan pronto como se detiene una iterable:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(map(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Transposición con mapa: uso de "Ninguno" como argumento de función (solo python 2.x)

```
from itertools import imap
from future_builtins import map as fmap # Different name to highlight differences

image = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]
```

```
list(map(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(fmap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(imap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

image2 = [[1, 2, 3],
          [4, 5],
          [7, 8, 9]]
list(map(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8), (3, None, 9)]  # Fill missing values with None
list(fmap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # ignore columns with missing values
list(imap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # dito
```

Python 3.x 3.0.0

```
list(map(None, *image))
```

TypeError: el objeto 'NoneType' no se puede llamar

Pero hay una solución para tener resultados similares:

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# Out: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Series y mapeo paralelo

map () es una función incorporada, lo que significa que está disponible en todas partes sin la necesidad de usar una declaración de 'importación'. Está disponible en todas partes, como print () Si observa el Ejemplo 5, verá que tuve que usar una declaración de importación antes de que pudiera usar una impresión bonita (pprint de importación). Así pprint no es una función incorporada

Mapeo en serie

En este caso, cada argumento de lo iterable se suministra como argumento a la función de mapeo en orden ascendente. Esto surge cuando solo tenemos un iterable para mapear y la función de mapeo requiere un solo argumento.

Ejemplo 1

```
insects = ['fly', 'ant', 'beetle', 'cankerworm']
f = lambda x: x + ' is an insect'
print(list(map(f, insects))) # the function defined by f is executed on each item of the
iterable insects
```

resultados en

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

Ejemplo 2

```
print(list(map(len, insects))) # the len function is executed each item in the insect list
```

resultados en

```
[3, 3, 6, 10]
```

Mapeo paralelo

En este caso, cada argumento de la función de mapeo se extrae de todos los iterables (uno de cada iterable) en paralelo. Por lo tanto, el número de iterables suministrados debe coincidir con el número de argumentos requeridos por la función.

```
carnivores = ['lion', 'tiger', 'leopard', 'arctic fox']
herbivores = ['african buffalo', 'moose', 'okapi', 'parakeet']
omnivores = ['chicken', 'dove', 'mouse', 'pig']

def animals(w, x, y, z):
    return '{0}, {1}, {2}, and {3} ARE ALL ANIMALS'.format(w.title(), x, y, z)
```

Ejemplo 3

```
# Too many arguments
# observe here that map is trying to pass one item each from each of the four iterables to
len. This leads len to complain that
# it is being fed too many arguments
print(list(map(len, insects, carnivores, herbivores, omnivores)))
```

resultados en

```
TypeError: len() takes exactly one argument (4 given)
```

Ejemplo 4

```
# Too few arguments
# observe here that map is suppose to execute animal on individual elements of insects one-by-
one. But animals complain when
# it only gets one argument, whereas it was expecting four.
print(list(map(animals, insects)))
```

resultados en

```
TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Ejemplo 5

```
# here map supplies w, x, y, z with one value from across the list
```

```
import pprint
pprint.pprint(list(map(animals, insects, carnivores, herbivores, omnivores)))
```

resultados en

```
 ['Fly, lion, african buffalo, and chicken ARE ALL ANIMALS',
 'Ant, tiger, moose, and dove ARE ALL ANIMALS',
 'Beetle, leopard, okapi, and mouse ARE ALL ANIMALS',
 'Cankerworm, arctic fox, parakeet, and pig ARE ALL ANIMALS']
```

Lea Función de mapa en línea: https://riptutorial.com/es/python/topic/333/funcion-de-mapa

Capítulo 88: Funciones

Introducción

Las funciones en Python proporcionan un código organizado, reutilizable y modular para realizar un conjunto de acciones específicas. Las funciones simplifican el proceso de codificación, evitan la lógica redundante y hacen que el código sea más fácil de seguir. Este tema describe la declaración y la utilización de funciones en Python.

Python tiene muchas *funciones integradas* como `print()` , `input()` , `len()` . Además de las funciones integradas, también puede crear sus propias funciones para realizar trabajos más específicos, que se denominan *funciones definidas por el usuario* .

Sintaxis

- def *function_name* (*arg1, ... argN, * args, kw1, kw2 = predeterminado, ..., ** kwargs*): *declaraciones*
- lambda *arg1, ... argN, * args, kw1, kw2 = predeterminado, ..., ** kwargs* : *expresión*

Parámetros

Parámetro	Detalles
arg1 , ..., *argN*	Argumentos regulares
* *args*	Argumentos posicionales sin nombre
kw1 , ..., *kwN*	Argumentos solo de palabra clave
** *kwargs*	El resto de argumentos de palabras clave.

Observaciones

5 cosas básicas que puedes hacer con las funciones:

- Asignar funciones a variables

```
def f():
  print(20)
y = f
y()
# Output: 20
```

- Definir funciones dentro de otras funciones (funciones anidadas)

```
def f(a, b, y):
    def inner_add(a, b):      # inner_add is hidden from outer code
        return a + b
    return inner_add(a, b)**y
```

- Las funciones pueden devolver otras funciones.

```
def f(y):
    def nth_power(x):
        return x ** y
    return nth_power    # returns a function

squareOf = f(2)         # function that returns the square of a number
cubeOf = f(3)           # function that returns the cube of a number
squareOf(3)             # Output: 9
cubeOf(2)               # Output: 8
```

- Las funciones se pueden pasar como parámetros a otras funciones.

```
def a(x, y):
    print(x, y)
def b(fun, str):        # b has two arguments: a function and a string
    fun('Hello', str)
b(a, 'Sophia')           # Output: Hello Sophia
```

- Las funciones internas tienen acceso al ámbito de cierre (Cierre)

```
def outer_fun(name):
    def inner_fun():     # the variable name is available to the inner function
        return "Hello "+ name + "!"
    return inner_fun
greet = outer_fun("Sophia")
print(greet())            # Output: Hello Sophia!
```

Recursos adicionales

- Más sobre funciones y decoradores: https://www.thecodeship.com/patterns/guide-to-python-function-decorators/

Examples

Definiendo y llamando funciones simples.

Usar la instrucción `def` es la forma más común de definir una función en python. Esta declaración es una *declaración compuesta* llamada *cláusula única* con la siguiente sintaxis:

```
def function_name(parameters):
    statement(s)
```

`function_name` se conoce como el *identificador* de la función. Dado que la definición de una función es una sentencia ejecutable, su ejecución *vincula* el nombre de la función con el objeto de la

función que se puede llamar más adelante utilizando el identificador.

`parameters` es una lista opcional de identificadores que se unen a los valores proporcionados como argumentos cuando se llama a la función. Una función puede tener un número arbitrario de argumentos separados por comas.

`statement(s)` también conocidas como el *cuerpo de* la *función*) son una secuencia no vacía de sentencias que se ejecutan cada vez que se llama a la función. Esto significa que el cuerpo de una función no puede estar vacío, como cualquier *bloque con sangría* .

Este es un ejemplo de una definición de función simple cuyo propósito es imprimir `Hello` cada vez que se llama:

```
def greet():
    print("Hello")
```

Ahora llamemos a la función de `greet()` definida:

```
greet()
# Out: Hello
```

Ese es otro ejemplo de una definición de función que toma un solo argumento y muestra el valor pasado cada vez que se llama a la función:

```
def greet_two(greeting):
    print(greeting)
```

Después de eso, la función `greet_two()` debe llamarse con un argumento:

```
greet_two("Howdy")
# Out: Howdy
```

También puede dar un valor predeterminado a ese argumento de función:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Ahora puedes llamar a la función sin dar un valor:

```
greet_two()
# Out: Howdy
```

Notará que, a diferencia de muchos otros idiomas, no necesita declarar explícitamente un tipo de devolución de la función. Las funciones de Python pueden devolver valores de cualquier tipo a través de la palabra clave `return` . ¡Una función puede devolver cualquier número de tipos diferentes!

```
def many_types(x):
    if x < 0:
```

```
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))

# Output:
0
Hello!
```

Mientras la persona que llama maneje esto correctamente, este es un código de Python perfectamente válido.

Una función que llega al final de la ejecución sin una declaración de retorno siempre devolverá `None` :

```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

Como se mencionó anteriormente, una definición de función debe tener un cuerpo de función, una secuencia de sentencias no vacía. Por lo tanto, la instrucción de `pass` se utiliza como cuerpo de la función, que es una operación nula: cuando se ejecuta, no sucede nada. Hace lo que significa, salta. Es útil como marcador de posición cuando se requiere una declaración sintácticamente, pero no es necesario ejecutar ningún código.

Devolviendo valores desde funciones

Las funciones pueden `return` un valor que puede utilizar directamente:

```
def give_me_five():
    return 5

print(give_me_five())  # Print the returned value
# Out: 5
```

o guarda el valor para su uso posterior:

```
num = give_me_five()
print(num)             # Print the saved returned value
# Out: 5
```

o use el valor para cualquier operación:

```
print(give_me_five() + 10)
# Out: 15
```

Si se encuentra `return` en la función, la función se cerrará inmediatamente y las operaciones

subsiguientes no se evaluarán:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

También puede `return` varios valores (en forma de tupla):

```
def give_me_two_fives():
    return 5, 5  # Returns two 5

first, second = give_me_two_fives()
print(first)
# Out: 5
print(second)
# Out: 5
```

Una función *sin una* declaración de `return` devuelve implícitamente `None` . De manera similar, una función con una declaración de `return` , pero sin valor de retorno o variable devuelve `None` .

Definiendo una función con argumentos.

Los argumentos se definen entre paréntesis después del nombre de la función:

```
def divide(dividend, divisor):  # The names of the function and its arguments
    # The arguments are available by name in the body of the function
    print(dividend / divisor)
```

El nombre de la función y su lista de argumentos se denominan la *firma* de la función. Cada argumento nombrado es efectivamente una variable local de la función.

Al llamar a la función, dé valores para los argumentos enumerándolos en orden

```
divide(10, 2)
# output: 5
```

o especifíquelos en cualquier orden usando los nombres de la definición de función:

```
divide(divisor=2, dividend=10)
# output: 5
```

Definiendo una función con argumentos opcionales.

Los argumentos opcionales se pueden definir asignando (usando `=`) un valor predeterminado al nombre de argumento:

```
def make(action='nothing'):
    return action
```

Llamar a esta función es posible de 3 maneras diferentes:

```
make("fun")
# Out: fun

make(action="sleep")
# Out: sleep

# The argument is optional so the function will use the default value if the argument is
# not passed in.
make()
# Out: nothing
```

Advertencia

Los tipos mutables (`list` , `dict` , `set` , etc.) deben tratarse con cuidado cuando se dan como atributo **predeterminado** . Cualquier mutación del argumento por defecto lo cambiará permanentemente. Consulte Definir una función con argumentos mutables opcionales .

Definiendo una función con múltiples argumentos.

Uno puede dar a la función tantos argumentos como quiera, las únicas reglas fijas son que cada nombre de argumento debe ser único y que los argumentos opcionales deben estar después de los no opcionales:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

Al llamar a la función, puede dar cada palabra clave sin el nombre, pero el orden importa:

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

O combinar dando los argumentos con nombre y sin. Entonces los que tienen nombre deben seguir a los que no tienen, pero el orden de los que tienen nombre no importa:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

Definiendo una función con un número arbitrario de argumentos.

Número arbitrario de argumentos

posicionales:

La definición de una función capaz de tomar un número arbitrario de argumentos se puede hacer prefijando uno de los argumentos con un `*`

```
def func(*args):
    # args will be a tuple containing all values that are passed in
    for i in args:
        print(i)

func(1, 2, 3)  # Calling it with 3 arguments
# Out: 1
#      2
#      3

list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values)  # Calling it with list of values, * expands the list
# Out: 1
#      2
#      3

func()  # Calling it without arguments
# No Output
```

No **puede** proporcionar un valor predeterminado para `args` , por ejemplo `func(*args=[1, 2, 3])` generará un error de sintaxis (ni siquiera compilará).

No **puede** proporcionarlos por nombre al llamar a la función, por ejemplo, `func(*args=[1, 2, 3])` generará un `TypeError` .

Pero si ya tiene sus argumentos en una matriz (o cualquier otro `Iterable`), **puede** invocar su función así: `func(*my_stuff)` .

Se puede acceder a estos argumentos (`*args`) por índice, por ejemplo, `args[0]` devolverá el primer argumento

Número arbitrario de argumentos de palabras clave

Puede tomar un número arbitrario de argumentos con un nombre definiendo un argumento en la definición con **dos** `*` delante:

```
def func(**kwargs):
    # kwargs will be a dictionary containing the names as keys and the values as values
    for name, value in kwargs.items():
        print(name, value)

func(value1=1, value2=2, value3=3)   # Calling it with 3 arguments
# Out: value1 1
#      value2 2
```

```
#      value3 3

func()                                  # Calling it without arguments
# No Out put

my_dict = {'foo': 1, 'bar': 2}
func(**my_dict)                         # Calling it with a dictionary
# Out: foo 1
#      bar 2
```

No **puede** proporcionarlos **sin** nombres, por ejemplo, `func(1, 2, 3)` generará un `TypeError` .

`kwargs` es un simple diccionario nativo de python. Por ejemplo, `args['value1']` dará el valor para el argumento `value1` . Asegúrese de verificar de antemano que exista tal argumento o se `KeyError` un `KeyError` .

Advertencia

Puede combinar estos con otros argumentos opcionales y requeridos, pero el orden dentro de la definición es importante.

Los argumentos **posicionales / palabras clave** son lo primero. (Argumentos requeridos).
Luego vienen los argumentos **arbitrarios** `*arg` . (Opcional).
Luego vienen los argumentos de **palabra clave** . (Necesario).
Finalmente la **palabra clave arbitraria** `**kwargs` viene. (Opcional).

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
     pass
```

- `arg1` debe darse, de lo contrario se `TypeError` un `TypeError` . Se puede dar como argumento posicional (`func(10)`) o de palabra clave (`func(arg1=10)`).
- `kwarg1` debe proporcionar `kwarg1` , pero solo se puede proporcionar como palabra clave-argumento: `func(kwarg1=10)` .
- `arg2` y `kwarg2` son opcionales. Si se va a cambiar el valor, se aplican las mismas reglas que para `arg1` (posicional o palabra clave) y `kwarg1` (solo palabra clave).
- `*args` captura parámetros posicionales adicionales. Pero tenga en cuenta que `arg1` y `arg2` deben proporcionarse como argumentos posicionales para pasar argumentos a `*args` : `func(1, 1, 1, 1)` .
- `**kwargs` captura todos los parámetros de palabras clave adicionales. En este caso, cualquier parámetro que no sea `arg1` , `arg2` , `kwarg1` o `kwarg2` . Por ejemplo: `func(kwarg3=10)` .
- En Python 3, puede usar `*` solo para indicar que todos los argumentos posteriores deben especificarse como palabras clave. Por ejemplo, la función `math.isclose` en Python 3.5 y superior se define usando `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)` , lo que significa que los primeros dos argumentos se pueden suministrar de manera posicional pero el opcional Los parámetros tercero y cuarto solo se pueden suministrar como argumentos de palabras clave.

Python 2.x no admite parámetros de palabras clave solamente. Este comportamiento puede ser emulado con `kwargs` :

```
def func(arg1, arg2=10, **kwargs):
    try:
        kwarg1 = kwargs.pop("kwarg1")
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # function body ...
```

Nota sobre nombrar

La convención de nombrar argumentos opcionales posicionales `args` y opcionales de palabras clave `kwargs` es solo una convención que **puede** usar cualquier nombre que desee, **pero** es útil seguir la convención para que otros sepan lo que está haciendo, *o incluso usted mismo más adelante,* así que hágalo.

Nota sobre la singularidad

Cualquier función se puede definir con **ninguno o un** `*args` y **ninguno o uno** `**kwargs` pero no con más de uno de cada uno. También `*args` **debe** ser el último argumento posicional y `**kwargs` debe ser el último parámetro. El intento de utilizar más de uno de ambos **dará** lugar a una excepción de error de sintaxis.

Nota sobre funciones de anidamiento con argumentos opcionales

Es posible anidar dichas funciones y la convención habitual es eliminar los elementos que el código ya ha manejado, **pero** si está pasando los parámetros, debe pasar argumentos opcionales posicionales con un prefijo `*` y argumentos de palabras clave opcionales con un prefijo `**` , de lo contrario, los argumentos se pueden pasar como una lista o tupla y los kwargs como un solo diccionario. p.ej:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)

def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

Definiendo una función con argumentos mutables opcionales.

Existe un problema cuando se usan **argumentos opcionales** con un **tipo predeterminado mutable** (descrito en Definición de una función con argumentos opcionales), lo que puede conducir a un comportamiento inesperado.

Explicación

Este problema surge porque los argumentos predeterminados de una función se inicializan **una vez** , en el momento en que se *define* la función y **no** (como en muchos otros idiomas) cuando se *llama a* la función. Los valores predeterminados se almacenan dentro de la variable miembro `__defaults__` del objeto de `__defaults__` .

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

Para los tipos **inmutables** (ver Pasar argumento y mutabilidad) esto no es un problema porque no hay manera de mutar la variable; solo se puede reasignar, sin cambiar el valor original. Por lo tanto, se garantiza que los subsiguientes tengan el mismo valor predeterminado. Sin embargo, para un tipo **mutable** , el valor original puede mutar, haciendo llamadas a sus diversas funciones miembro. Por lo tanto, no se garantiza que las llamadas sucesivas a la función tengan el valor predeterminado inicial.

```
def append(elem, to=[]):
    to.append(elem)      # This call to append() mutates the default variable "to"
    return to

append(1)
# Out: [1]

append(2)  # Appends it to the internally stored list
# Out: [1, 2]

append(3, [])  # Using a new created list gives the expected result
# Out: [3]

# Calling it again without argument will append to the internally stored list again
append(4)
# Out: [1, 2, 4]
```

> **Nota:** Algunos IDE como PyCharm emitirán una advertencia cuando se especifique un tipo mutable como atributo predeterminado.

Solución

Si desea asegurarse de que el argumento predeterminado sea siempre el que especifique en la definición de la función, entonces la solución es usar **siempre** un tipo inmutable como su argumento predeterminado.

Un modismo común para lograr esto cuando se necesita un tipo mutable como predeterminado, es utilizar `None` (inmutable) como argumento predeterminado y luego asignar el valor predeterminado real a la variable de argumento si es igual a `None` .

```
def append(elem, to=None):
    if to is None:
        to = []

    to.append(elem)
    return to
```

Funciones Lambda (Inline / Anónimo)

La palabra clave `lambda` crea una función en línea que contiene una sola expresión. El valor de esta expresión es lo que la función devuelve cuando se invoca.

Considere la función:

```
def greeting():
    return "Hello"
```

el cual, cuando es llamado como:

```
print(greeting())
```

huellas dactilares:

```
Hello
```

Esto se puede escribir como una función lambda de la siguiente manera:

```
greet_me = lambda: "Hello"
```

> Vea la nota al final de esta sección con respecto a la asignación de lambdas a las variables. En general, no lo hagas.

Esto crea una función en línea con el nombre `greet_me` que devuelve `Hello` . Tenga en cuenta que no escribe `return` cuando crea una función con lambda. El valor después de `:` se devuelve automáticamente.

Una vez asignada a una variable, se puede usar como una función regular:

```
print(greet_me())
```

huellas dactilares:

```
Hello
```

`lambda`

s puede tomar argumentos, también:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

devuelve la cadena:

```
HELLO
```

También pueden tomar un número arbitrario de argumentos / argumentos de palabras clave, como las funciones normales.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

huellas dactilares:

```
hello ('world',) {'world': 'world'}
```

`lambda` se usan comúnmente para funciones cortas que son convenientes para definir en el punto donde se llaman (típicamente con `sorted` , `filter` y `map`).

Por ejemplo, esta línea ordena una lista de cadenas que ignoran su caso e ignoran los espacios en blanco al principio y al final:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip().upper())
# Out:
# ['    bAR', 'BaZ    ', ' foo ']
```

Ordenar la lista simplemente ignorando espacios en blanco:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip())
# Out:
# ['BaZ    ', '    bAR', ' foo ']
```

Ejemplos con `map` :

```
sorted( map( lambda s: s.strip().upper(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BAR', 'BAZ', 'FOO']

sorted( map( lambda s: s.strip(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BaZ', 'bAR', 'foo']
```

Ejemplos con listas numéricas:

```
my_list = [3, -4, -2, 5, 1, 7]
sorted( my_list, key=lambda x: abs(x))
```

```
# Out:
# [1, -2, 3, -4, 5, 7]

list( filter( lambda x: x>0, my_list))
# Out:
# [3, 5, 1, 7]

list( map( lambda x: abs(x), my_list))
# Out:
[3, 4, 2, 5, 1, 7]
```

Uno puede llamar a otras funciones (con / sin argumentos) desde dentro de una función lambda.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

huellas dactilares:

```
hello world
```

Esto es útil porque `lambda` puede contener solo una expresión y al usar una función subsidiaria se pueden ejecutar varias declaraciones.

NOTA

Tenga en cuenta que PEP-8 (la guía de estilo oficial de Python) no recomienda asignar lambdas a las variables (como hicimos en los dos primeros ejemplos):

> Siempre use una instrucción def en lugar de una instrucción de asignación que vincule una expresión lambda directamente a un identificador.
>
> Sí:
>
> ```
> def f(x): return 2*x
> ```
>
> No:
>
> ```
> f = lambda x: 2*x
> ```
>
> La primera forma significa que el nombre del objeto de función resultante es específicamente `f` lugar del genérico `<lambda>` . Esto es más útil para las trazas y representaciones de cadenas en general. El uso de la declaración de asignación elimina el único beneficio que puede ofrecer una expresión lambda sobre una declaración explícita de `def` (es decir, que se puede incrustar dentro de una expresión más grande).

Argumento de paso y mutabilidad.

Primero, alguna terminología:

- **argumento (parámetro *real*):** la variable real que se pasa a una función;
- **parámetro (parámetro *formal*):** la variable de recepción que se utiliza en una función.

En Python, los argumentos se pasan por *asignación* (a diferencia de otros idiomas, donde los argumentos pueden pasarse por valor / referencia / puntero).

- Mutar un parámetro mutará el argumento (si el tipo del argumento es mutable).

```
def foo(x):         # here x is the parameter
    x[0] = 9        # This mutates the list labelled by both x and y
    print(x)

y = [4, 5, 6]
foo(y)              # call foo with y as argument
# Out: [9, 5, 6]   # list labelled by x has been mutated
print(y)
# Out: [9, 5, 6]   # list labelled by y has been mutated too
```

- Reasignar el parámetro no reasignará el argumento.

```
def foo(x):         # here x is the parameter, when we call foo(y) we assign y to x
    x[0] = 9        # This mutates the list labelled by both x and y
    x = [1, 2, 3]  # x is now labeling a different list (y is unaffected)
    x[2] = 8        # This mutates x's list, not y's list

y = [4, 5, 6]       # y is the argument, x is the parameter
foo(y)              # Pretend that we wrote "x = y", then go to line 1
y
# Out: [9, 5, 6]
```

En Python, realmente no asignamos valores a las variables, en lugar de eso, *vinculamos* (es decir, asignamos, adjuntamos) variables (consideradas como *nombres*) a los objetos.

- **Inmutable:** enteros, cadenas, tuplas, etc. Todas las operaciones hacen copias.
- **Mutable:** listas, diccionarios, conjuntos, etc. Las operaciones pueden o no mutar.

```
x = [3, 1, 9]
y = x
x.append(5)     # Mutates the list labelled by x and y, both x and y are bound to [3, 1, 9]
x.sort()        # Mutates the list labelled by x and y (in-place sorting)
x = x + [4]     # Does not mutate the list (makes a copy for x only, not y)
z = x           # z is x ([1, 3, 9, 4])
x += [6]        # Mutates the list labelled by both x and z (uses the extend function).
x = sorted(x)   # Does not mutate the list (makes a copy for x only).
x
# Out: [1, 3, 4, 5, 6, 9]
y
# Out: [1, 3, 5, 9]
z
# Out: [1, 3, 5, 9, 4, 6]
```

Cierre

Los cierres en Python son creados por llamadas a funciones. Aquí, la llamada a `makeInc` crea un enlace para `x` que se hace referencia dentro de la función `inc` . Cada llamada a `makeInc` crea una nueva instancia de esta función, pero cada instancia tiene un enlace a un enlace diferente de `x` .

```
def makeInc(x):
  def inc(y):
     # x is "attached" in the definition of inc
     return y + x

  return inc

incOne = makeInc(1)
incFive = makeInc(5)

incOne(5) # returns 6
incFive(5) # returns 10
```

Observe que, mientras que en un cierre regular, la función encerrada hereda completamente todas las variables de su entorno envolvente, en esta construcción, la función encerrada solo tiene acceso de lectura a las variables heredadas, pero no puede asignárselas.

```
def makeInc(x):
  def inc(y):
     # incrementing x is not allowed
     x += y
     return x

  return inc

incOne = makeInc(1)
incOne(5) # UnboundLocalError: local variable 'x' referenced before assignment
```

Python 3 ofrece la declaración no `nonlocal` (variables no locales) para realizar un cierre completo con funciones anidadas.

Python 3.x 3.0

```
def makeInc(x):
  def inc(y):
     nonlocal x
     # now assigning a value to x is allowed
     x += y
     return x

  return inc

incOne = makeInc(1)
incOne(5) # returns 6
```

Funciones recursivas

Una función recursiva es una función que se llama a sí misma en su definición. Por ejemplo, la

función matemática, factorial, definida por `factorial(n) = n*(n-1)*(n-2)*...*3*2*1` . se puede programar como

```
def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Las salidas aquí son:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

como se esperaba. Observe que esta función es recursiva porque el segundo `return factorial(n-1)` , donde la función se llama a sí mismo en su definición.

Algunas funciones recursivas pueden implementarse usando lambda , la función factorial que usa lambda sería algo como esto:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

La función produce la misma salida que la anterior.

Límite de recursión

Hay un límite a la profundidad de la posible recursión, que depende de la implementación de Python. Cuando se alcanza el límite, se genera una excepción RuntimeError:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))

cursing(0)
# Out: I recursed 1083 times!
```

Es posible cambiar el límite de profundidad de recursión usando `sys.setrecursionlimit(limit)` y verificar este límite por `sys.getrecursionlimit()` .

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

Desde Python 3.5, la excepción es un `RecursionError` , que se deriva de `RuntimeError` .

Funciones anidadas

Las funciones en python son objetos de primera clase. Se pueden definir en cualquier ámbito.

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
    return a
```

Las funciones que capturan su ámbito de distribución pueden transmitirse como cualquier otro tipo de objeto

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Out: 15
add6(10)
#Out: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

Iterable y desempaquetado del diccionario.

Las funciones le permiten especificar estos tipos de parámetros: posicionales, nombrados, variables posicionales, Argumentos de palabras clave (kwargs). Aquí hay un uso claro y conciso de cada tipo.

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
    print(a, b, c, d, args, kwargs)

>>> unpacking(1, 2)
1 2 45 60 () {}
>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}
```

```
>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> args_list = [3]
>>> unpacking(1, 2, *args_list, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, *args_list, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'


>>> pair = (3, 4)
>>> unpacking(1, 2, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *pair)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *pair, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'



>>> args_list = [3, 4]
>>> unpacking(1, 2, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *args_list)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *args_list, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'


>>> arg_dict = {'c':3, 'd':4}
```

```
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'d':4, 'c':3}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'c':3, 'd':4, 'not_a_parameter': 75}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {'not_a_parameter': 75}


>>> unpacking(1, 2, *pair, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, 3, 4, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

# Positional arguments take priority over any other form of argument passing
>>> unpacking(1, 2, **arg_dict, c=3)
1 2 3 4 () {'not_a_parameter': 75}
>>> unpacking(1, 2, 3, **arg_dict, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
```

Forzando el uso de parámetros nombrados

Todos los parámetros especificados después del primer asterisco en la firma de función son solo palabras clave.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

En Python 3 es posible poner un solo asterisco en la firma de la función para garantizar que los argumentos restantes solo puedan pasarse utilizando argumentos de palabras clave.

```
def f(a, b, *, c):
    pass

f(1, 2, 3)
# TypeError: f() takes 2 positional arguments but 3 were given
f(1, 2, c=3)
# No error
```

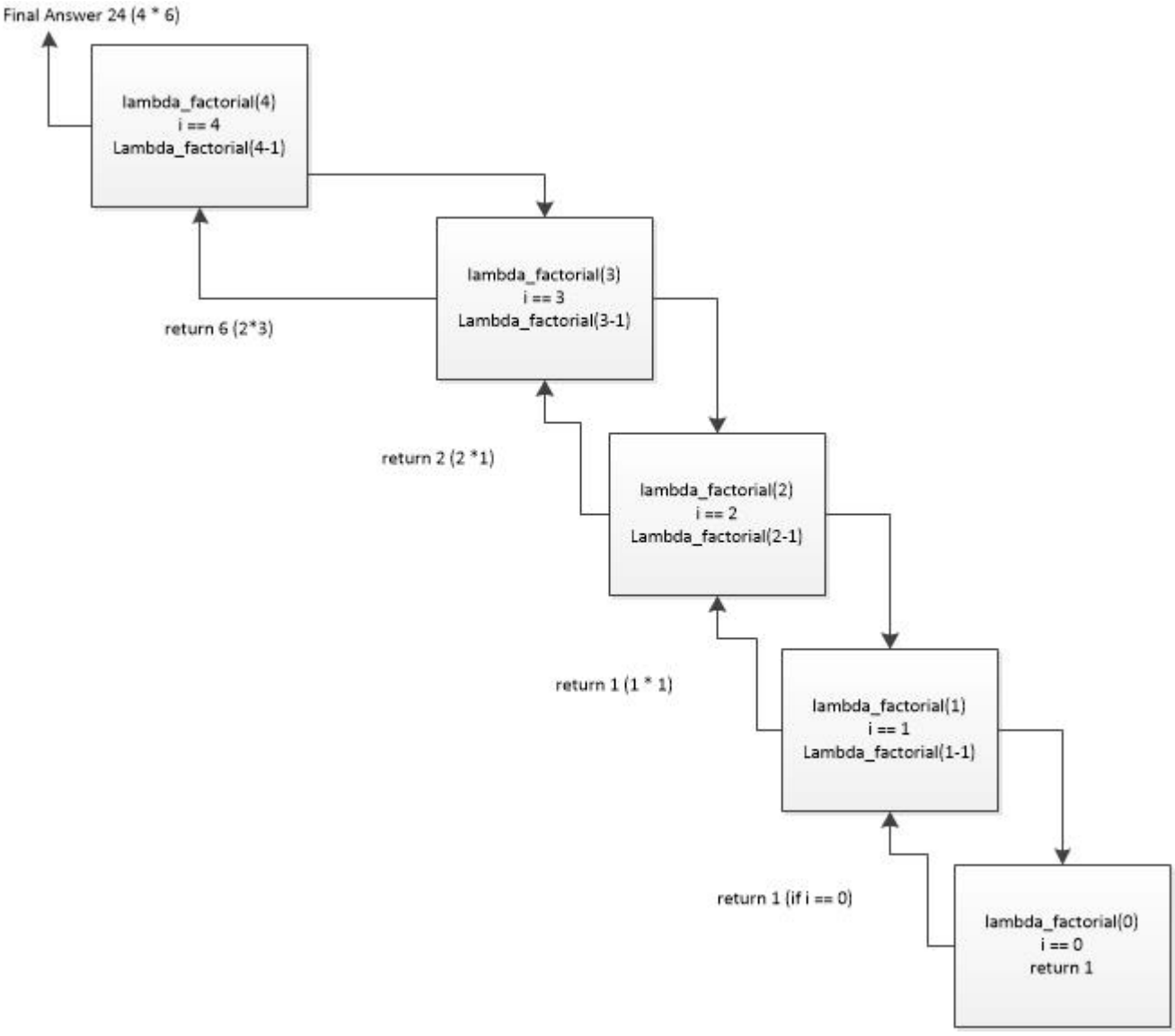
Lambda recursiva utilizando variable asignada

Un método para crear funciones lambda recursivas implica asignar la función a una variable y luego hacer referencia a esa variable dentro de la misma función. Un ejemplo común de esto es el cálculo recursivo del factorial de un número, como se muestra en el siguiente código:

```
lambda_factorial = lambda i:1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24
```

Descripción del código

A la función lambda, a través de su asignación variable, se le pasa un valor (4) que evalúa y devuelve 1 si es 0 o, de lo contrario, devuelve el valor actual (`i`) * otro cálculo realizado por la función lambda del valor - 1 (`i-1`). Esto continúa hasta que el valor pasado se reduce a 0 (`return 1`). Un proceso que se puede visualizar como:



Lea Funciones en línea: https://riptutorial.com/es/python/topic/228/funciones

Capítulo 89: Funciones parciales

Introducción

Como probablemente sepa si vino de la escuela de OOP, especializarse en una clase abstracta y usarla es una práctica que debe tener en cuenta al escribir su código.

¿Qué pasaría si pudieras definir una función abstracta y especializarla para crear diferentes versiones de ella? Piensa que es una especie de *función de herencia* donde se vinculan parámetros específicos para que sean confiables para un escenario específico.

Sintaxis

- parcial (función, ** params_you_want_fix)

Parámetros

Param	detalles
X	el número a elevar
y	el exponente
aumento	La función a especializarse.

Observaciones

Como se indica en el documento de Python, *functools.partial* :

> Devuelva un nuevo objeto parcial que, cuando se le llame, se comportará como una función llamada con los argumentos posicionales, argumentos y palabras clave de palabras clave. Si se proporcionan más argumentos a la llamada, se anexan a args. Si se proporcionan argumentos de palabras clave adicionales, amplían y anulan las palabras clave.

Consulte este enlace para ver cómo se puede implementar *parcial* .

Examples

Elevar el poder

Supongamos que queremos elevar *x* a un número *y* .

Escribirías esto como:

```
def raise_power(x, y):
    return x**y
```

¿Qué pasa si su valor *y* puede asumir un conjunto finito de valores?

Supongamos que *y* puede ser uno de *[3,4,5]* y digamos que no quiere ofrecer al usuario final la posibilidad de usar dicha función, ya que es muy computacional. De hecho, verificará si se proporciona *y* asume un valor válido y reescribirá su función como:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

¿Sucio? Usemos el formulario abstracto y lo especialicemos en los tres casos: implementémoslo **parcialmente** .

```
from functors import partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

¿Qué pasa aquí? Hemos fijado los parametros *y* y definimos tres funciones diferentes.

No es necesario usar la función abstracta definida anteriormente (puede hacerla *privada*), pero puede usar funciones **aplicadas parciales** para tratar de elevar un número a un valor fijo.

Lea Funciones parciales en línea: https://riptutorial.com/es/python/topic/9383/funciones-parciales

Capítulo 90: Generadores

Introducción

Los generadores son iteradores perezosos creados por las funciones del generador (que utilizan el `yield`) o las expresiones del generador (que usan `(an_expression for x in an_iterator)`).

Sintaxis

- rendimiento `<expr>`
- rendimiento de `<expr>`
- `<var>` = rendimiento `<expr>`
- siguiente (`<iter>`)

Examples

Iteración

Un objeto generador soporta el *protocolo iterador* . Es decir, proporciona un método `next()` (`__next__()` en Python 3.x), que se utiliza para avanzar en su ejecución, y su método `__iter__` devuelve. Esto significa que se puede usar un generador en cualquier construcción de lenguaje que admita objetos iterables genéricos.

```
# naive partial implementation of the Python 2.x xrange()
def xrange(n):
    i = 0
    while i < n:
        yield i
        i += 1

# looping
for i in xrange(10):
    print(i)  # prints the values 0, 1, ..., 9

# unpacking
a, b, c = xrange(3)  # 0, 1, 2

# building a list
l = list(xrange(10))  # [0, 1, ..., 9]
```

La siguiente función ()

El `next()` incorporado es un envoltorio conveniente que se puede usar para recibir un valor de cualquier iterador (incluido un iterador generador) y para proporcionar un valor predeterminado en caso de que se agote el iterador.

```
def nums():
```

```
    yield 1
    yield 2
    yield 3
generator = nums()

next(generator, None)  # 1
next(generator, None)  # 2
next(generator, None)  # 3
next(generator, None)  # None
next(generator, None)  # None
# ...
```

La sintaxis es la `next(iterator[, default])` . Si el iterador finaliza y se pasa un valor predeterminado, se devuelve. Si no se proporcionó ningún valor predeterminado, se `StopIteration` .

Enviando objetos a un generador.

Además de recibir valores de un generador, es posible *enviar* un objeto a un generador utilizando el método `send()` .

```
def accumulator():
    total = 0
    value = None
    while True:
        # receive sent value
        value = yield total
        if value is None: break
        # aggregate values
        total += value

generator = accumulator()

# advance until the first "yield"
next(generator)      # 0

# from this point on, the generator aggregates values
generator.send(1)    # 1
generator.send(10)   # 11
generator.send(100)  # 111
# ...

# Calling next(generator) is equivalent to calling generator.send(None)
next(generator)      # StopIteration
```

Lo que pasa aquí es lo siguiente:

- Cuando llama por primera vez al `next(generator)` , el programa avanza a la primera declaración de `yield` y devuelve el valor del `total` en ese punto, que es 0. La ejecución del generador se suspende en este punto.
- Cuando llama a `generator.send(x)` , el intérprete toma el argumento `x` y lo convierte en el valor de retorno de la última declaración de `yield` , que se asigna al `value` . El generador continúa como de costumbre, hasta que da el siguiente valor.
- Cuando finalmente llama a `next(generator)` , el programa trata esto como si estuviera

enviando `None` al generador. No hay nada especial en `None` , sin embargo, este ejemplo utiliza `None` como un valor especial para pedirle al generador que se detenga.

Expresiones generadoras

Es posible crear iteradores de generador utilizando una sintaxis similar a la comprensión.

```
generator = (i * 2 for i in range(3))

next(generator)  # 0
next(generator)  # 2
next(generator)  # 4
next(generator)  # raises StopIteration
```

Si a una función no necesariamente se le debe pasar una lista, puede guardar caracteres (y mejorar la legibilidad) colocando una expresión de generador dentro de una llamada de función. El paréntesis de la llamada a la función hace implícitamente que su expresión sea una expresión generadora.

```
sum(i ** 2 for i in range(4))  # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

Además, guardará en la memoria porque en lugar de cargar la lista completa sobre la que está iterando (`[0, 1, 2, 3]` en el ejemplo anterior), el generador permite que Python use los valores según sea necesario.

Introducción

Las expresiones de los generadores son similares a las listas, diccionarios y conjuntos de comprensión, pero están entre paréntesis. Los paréntesis no tienen que estar presentes cuando se usan como el único argumento para una llamada de función.

```
expression = (x**2 for x in range(10))
```

Este ejemplo genera los 10 primeros cuadrados perfectos, incluido 0 (en el que x = 0).

Las funciones del generador son similares a las funciones regulares, excepto que tienen una o más declaraciones de `yield` en su cuerpo. Dichas funciones no pueden `return` ningún valor (sin embargo, se permiten `return` vacías si desea detener el generador antes).

```
def function():
    for x in range(10):
        yield x**2
```

Esta función del generador es equivalente a la expresión del generador anterior, produce el mismo.

Nota : todas las expresiones generadoras tienen sus propias funciones *equivalentes* , pero no al revés.

Se puede usar una expresión generadora sin paréntesis si ambos paréntesis se repetirían de lo contrario:

```
sum(i for i in range(10) if i % 2 == 0)   #Output: 20
any(x = 0 for x in foo)                   #Output: True or False depending on foo
type(a > b for a in foo if a % 2 == 1)    #Output: <class 'generator'>
```

En lugar de:

```
sum((i for i in range(10) if i % 2 == 0))
any((x = 0 for x in foo))
type((a > b for a in foo if a % 2 == 1))
```

Pero no:

```
fooFunction(i for i in range(10) if i % 2 == 0,foo,bar)
return x = 0 for x in foo
barFunction(baz, a > b for a in foo if a % 2 == 1)
```

Al llamar a una función de generador se genera un **objeto generador** , que luego se puede iterar. A diferencia de otros tipos de iteradores, los objetos generadores solo se pueden atravesar una vez.

```
g1 = function()
print(g1)  # Out: <generator object function at 0x1012e1888>
```

Observe que el cuerpo de un generador **no** se ejecuta inmediatamente: cuando llama a `function()` en el ejemplo anterior, devuelve inmediatamente un objeto generador, sin ejecutar siquiera la primera declaración de impresión. Esto permite que los generadores consuman menos memoria que las funciones que devuelven una lista, y permite crear generadores que producen secuencias infinitamente largas.

Por esta razón, los generadores a menudo se utilizan en la ciencia de datos y en otros contextos que involucran grandes cantidades de datos. Otra ventaja es que otro código puede usar inmediatamente los valores generados por un generador, sin esperar a que se produzca la secuencia completa.

Sin embargo, si necesita usar los valores producidos por un generador más de una vez, y si generarlos cuesta más que almacenarlos, puede ser mejor almacenar los valores generados como una `list` que volver a generar la secuencia. Consulte 'Restablecer un generador' a continuación para obtener más detalles.

Normalmente, un objeto generador se usa en un bucle, o en cualquier función que requiera un iterable:

```
for x in g1:
    print("Received", x)

# Output:
```

```
# Received 0
# Received 1
# Received 4
# Received 9
# Received 16
# Received 25
# Received 36
# Received 49
# Received 64
# Received 81

arr1 = list(g1)
# arr1 = [], because the loop above already consumed all the values.
g2 = function()
arr2 = list(g2)  # arr2 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Como los objetos generadores son iteradores, uno puede recorrerlos manualmente usando la función `next()` . Al hacerlo, se devolverán los valores cedidos uno por uno en cada invocación posterior.

Bajo el capó, cada vez que llama a `next()` en un generador, Python ejecuta declaraciones en el cuerpo de la función del generador hasta que llega a la siguiente declaración de `yield` . En este punto, devuelve el argumento del comando de `yield` y recuerda el punto en el que ocurrió. Llamar a `next()` una vez más reanudará la ejecución desde ese punto y continuará hasta la próxima declaración de `yield` .

Si Python llega al final de la función del generador sin encontrar más `yield` , se `StopIteration` una excepción `StopIteration` (esto es normal, todos los iteradores se comportan de la misma manera).

```
g3 = function()
a = next(g3)  # a becomes 0
b = next(g3)  # b becomes 1
c = next(g3)  # c becomes 2
...
j = next(g3)  # Raises StopIteration, j remains undefined
```

Tenga en cuenta que en el generador Python 2, los objetos tenían métodos `.next()` que se podían usar para iterar a través de los valores producidos manualmente. En Python 3, este método fue reemplazado por el estándar `.__next__()` para todos los iteradores.

Restablecer un generador

Recuerde que solo puede recorrer los objetos generados por un generador *una vez* . Si ya ha iterado a través de los objetos en una secuencia de comandos, cualquier otro intento de hacerlo dará como resultado `None` .

Si necesita usar los objetos generados por un generador más de una vez, puede definir la función del generador de nuevo y usarla por segunda vez, o bien, puede almacenar la salida de la función del generador en una lista en el primer uso. Volver a definir la función del generador será una buena opción si está manejando grandes volúmenes de datos, y almacenar una lista de todos los elementos de datos ocuparía mucho espacio en el disco. A la inversa, si es costoso generar los artículos inicialmente, es posible que prefiera almacenar los artículos generados en una lista para

poder reutilizarlos.

Usando un generador para encontrar los números de Fibonacci

Un caso de uso práctico de un generador es recorrer los valores de una serie infinita. Aquí hay un ejemplo de cómo encontrar los primeros diez términos de la secuencia de Fibonacci .

```
def fib(a=0, b=1):
    """Generator that yields Fibonacci numbers. `a` and `b` are the seed values"""
    while True:
        yield a
        a, b = b, a + b

f = fib()
print(', '.join(str(next(f)) for _ in range(10)))
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Secuencias infinitas

Se pueden usar generadores para representar secuencias infinitas:

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1

natural_numbers = integers_starting_from(1)
```

La secuencia infinita de números como la anterior también se puede generar con la ayuda de `itertools.count` . El código anterior se puede escribir como abajo

```
natural_numbers = itertools.count(1)
```

Puede usar la comprensión del generador en generadores infinitos para producir nuevos generadores:

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

Tenga en cuenta que un generador infinito no tiene un fin, por lo que pasarlo a cualquier función que intente consumir el generador por completo tendrá **graves consecuencias** :

```
list(multiples_of_two)  # will never terminate, or raise an OS-specific error
```

En su lugar, use la lista de listas / conjuntos con `range` (o `xrange` para python <3.0):

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

o use `itertools.islice()` para cortar el iterador en un subconjunto:

```
from itertools import islice
multiples_of_four = (x * 4 for x in integers_starting_from(1))
first_five_multiples_of_four = list(islice(multiples_of_four, 5))
# [4, 8, 12, 16, 20]
```

Tenga en cuenta que el generador original también se actualiza, al igual que todos los demás generadores que vienen de la misma "raíz":

```
next(natural_numbers)    # yields 16
next(multiples_of_two)   # yields 34
next(multiples_of_four)  # yields 24
```

Una secuencia infinita también se puede iterar con un bucle `for` . Asegúrese de incluir una instrucción de `break` condicional para que el bucle finalice eventualmente:

```
for idx, number in enumerate(multiplies_of_two):
    print(number)
    if idx == 9:
        break  # stop after taking the first 10 multiplies of two
```

Ejemplo clásico - números de Fibonacci

```
import itertools

def fibonacci():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b

first_ten_fibs = list(itertools.islice(fibonacci(), 10))
# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

def nth_fib(n):
    return next(itertools.islice(fibonacci(), n - 1, n))

ninety_nineth_fib = nth_fib(99)  # 354224848179261915075
```

Rindiendo todos los valores de otro iterable.

Python 3.x 3.3

Utilice el `yield from` si desea obtener todos los valores de otro iterable:

```
def foob(x):
    yield from range(x * 2)
    yield from range(2)

list(foob(5))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

Esto funciona también con generadores.

```
def fibto(n):
    a, b = 1, 1
    while True:
        if a >= n: break
        yield a
        a, b = b, a + b

def usefib():
    yield from fibto(10)
    yield from fibto(20)

list(usefib())  # [1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8, 13]
```

Coroutines

Los generadores pueden ser utilizados para implementar coroutines:

```
# create and advance generator to the first yield
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return start

# example coroutine
@coroutine
def adder(sum = 0):
    while True:
        x = yield sum
        sum += x

# example use
s = adder()
s.send(1) # 1
s.send(2) # 3
```

Coroutines se usa comúnmente para implementar máquinas de estado, ya que son principalmente útiles para crear procedimientos de un solo método que requieren un estado para funcionar correctamente. Operan en un estado existente y devuelven el valor obtenido al finalizar la operación.

Rendimiento con recursión: listado recursivo de todos los archivos en un directorio

Primero, importa las bibliotecas que trabajan con archivos:

```
from os import listdir
from os.path import isfile, join, exists
```

Una función auxiliar para leer solo archivos de un directorio:

```
def get_files(path):
    for file in listdir(path):
        full_path = join(path, file)
        if isfile(full_path):
            if exists(full_path):
                yield full_path
```

Otra función auxiliar para obtener solo los subdirectorios:

```
def get_directories(path):
    for directory in listdir(path):
        full_path = join(path, directory)
        if not isfile(full_path):
            if exists(full_path):
                yield full_path
```

Ahora use estas funciones para obtener recursivamente todos los archivos dentro de un directorio y todos sus subdirectorios (usando generadores):

```
def get_files_recursive(directory):
    for file in get_files(directory):
        yield file
    for subdirectory in get_directories(directory):
        for file in get_files_recursive(subdirectory): # here the recursive call
            yield file
```

Esta función se puede simplificar utilizando el `yield from` :

```
def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)
```

Iterando sobre generadores en paralelo.

Para iterar sobre varios generadores en paralelo, use el `zip` incorporado:

```
for x, y in zip(a,b):
    print(x,y)
```

Resultados en:

```
1 x
2 y
3 z
```

En Python 2 deberías usar `itertools.izip` en `itertools.izip` lugar. Aquí también podemos ver que todas las funciones `zip` producen tuplas.

Tenga en cuenta que zip dejará de iterar tan pronto como uno de los iterables se quede sin elementos. Si desea iterar durante el tiempo más largo posible, use `itertools.zip_longest()` .

Código de construcción de lista de refactorización

Supongamos que tiene un código complejo que crea y devuelve una lista al comenzar con una lista en blanco y agregarla repetidamente:

```
def create():
    result = []
    # logic here...
    result.append(value) # possibly in several places
    # more logic...
    return result # possibly in several places

values = create()
```

Cuando no es práctico reemplazar la lógica interna con una lista de comprensión, puede convertir toda la función en un generador en el lugar y luego recopilar los resultados:

```
def create_gen():
    # logic...
    yield value
    # more logic
    return # not needed if at the end of the function, of course

values = list(create_gen())
```

Si la lógica es recursiva, use el `yield from` para incluir todos los valores de la llamada recursiva en un resultado "aplanado":

```
def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)
```

buscando

La `next` función es útil incluso sin iterar. Pasar una expresión del generador a la `next` es una forma rápida de buscar la primera aparición de un elemento que coincida con algún predicado. Código de procedimiento como

```
def find_and_transform(sequence, predicate, func):
    for element in sequence:
        if predicate(element):
            return func(element)
    raise ValueError

item = find_and_transform(my_sequence, my_predicate, my_func)
```

puede ser reemplazado con:

```
item = next(my_func(x) for x in my_sequence if my_predicate(x))
# StopIteration will be raised if there are no matches; this exception can
# be caught and transformed, if desired.
```

Para este propósito, puede ser conveniente crear un alias, como `first = next` , o una función de envoltura para convertir la excepción:

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

Lea Generadores en línea: https://riptutorial.com/es/python/topic/292/generadores

Capítulo 91: Gestores de contexto (declaración “con”)

Introducción

Si bien los administradores de contexto de Python son ampliamente utilizados, pocos entienden el propósito detrás de su uso. Estas declaraciones, comúnmente utilizadas con los archivos de lectura y escritura, ayudan a la aplicación a conservar la memoria del sistema y mejorar la administración de recursos al garantizar que los recursos específicos solo se usan para ciertos procesos. Este tema explica y demuestra el uso de los gestores de contexto de Python.

Sintaxis

- con "context_manager" (como "alias") (, "context_manager" (como "alias")?) *:

Observaciones

Los gestores de contexto se definen en PEP 343 . Están diseñados para ser utilizados como un mecanismo más sucinto para la administración de recursos que el que `try ... finally` construye. La definición formal es la siguiente.

> En este PEP, los gestores de contexto proporcionan los `__enter__()` y `__exit__()` que se invocan al ingresar y salir del cuerpo de la declaración with.

Luego pasa a definir la instrucción `with` lo siguiente.

```
with EXPR as VAR:
    BLOCK
```

La traducción de la declaración anterior es:

```
   mgr = (EXPR)
   exit = type(mgr).__exit__  # Not calling it yet
   value = type(mgr).__enter__(mgr)
   exc = True
   try:
       try:
           VAR = value  # Only if "as VAR" is present
           BLOCK
       except:
           # The exceptional case is handled here
           exc = False
           if not exit(mgr, *sys.exc_info()):
               raise
           # The exception is swallowed if exit() returns true
   finally:
       # The normal and non-local-goto cases are handled here
```

```
        if exc:
            exit(mgr, None, None, None)
```

Examples

Introducción a los gestores de contexto y con la declaración.

Un administrador de contexto es un objeto que se notifica cuando un contexto (un bloque de código) *comienza* y *termina* . Normalmente se utiliza uno con la instrucción `with` . Se encarga de la notificación.

Por ejemplo, los objetos de archivo son gestores de contexto. Cuando un contexto finaliza, el objeto de archivo se cierra automáticamente:

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

El ejemplo anterior generalmente se simplifica utilizando la palabra clave `as` :

```
with open(filename) as open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Cualquier cosa que finalice la ejecución del bloque hace que se llame al método de salida del administrador de contexto. Esto incluye excepciones y puede ser útil cuando un error hace que salga prematuramente de un archivo abierto o conexión. Salir de un script sin cerrar correctamente los archivos / conexiones es una mala idea, ya que puede causar la pérdida de datos u otros problemas. Al utilizar un administrador de contexto, puede asegurarse de que siempre se tomen precauciones para evitar daños o pérdidas de esta manera. Esta característica fue añadida en Python 2.5.

Asignar a un objetivo

Muchos administradores de contexto devuelven un objeto cuando se ingresa. Puede asignar ese objeto a un nuevo nombre en la declaración `with` .

Por ejemplo, usar una conexión de base de datos en una declaración `with` podría darle un objeto de cursor:

```
with database_connection as cursor:
    cursor.execute(sql_query)
```

Los objetos de archivo se devuelven solos, esto hace posible abrir el objeto de archivo y usarlo como administrador de contexto en una expresión:

```
with open(filename) as open_file:
    file_contents = open_file.read()
```

Escribiendo tu propio gestor de contexto.

Un administrador de contexto es cualquier objeto que implementa dos métodos mágicos `__enter__()` y `__exit__()` (aunque también puede implementar otros métodos):

```
class AContextManager():

    def __enter__(self):
        print("Entered")
        # optionally return an object
        return "A-instance"

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exited" + (" (with an exception)" if exc_type else ""))
        # return True if you want to suppress the exception
```

Si el contexto sale con una excepción, la información sobre esa excepción se pasa como un triple `exc_type` , `exc_value` , `traceback` (estas son las mismas variables que devuelve el `sys.exc_info()` función). Si el contexto sale normalmente, los tres de estos argumentos serán `None` .

Si ocurre una excepción y se pasa al método `__exit__` , el método puede devolver `True` para suprimir la excepción, o la excepción volverá a presentarse al final de la función `__exit__` .

```
with AContextManager() as a:
    print("a is %r" % a)
# Entered
# a is 'A-instance'
# Exited

with AContextManager() as a:
    print("a is %d" % a)
# Entered
# Exited (with an exception)
# Traceback (most recent call last):
#   File "<stdin>", line 2, in <module>
# TypeError: %d format: a number is required, not str
```

Tenga en cuenta que en el segundo ejemplo, a pesar de que se produce una excepción en medio del cuerpo de la instrucción with, el controlador `__exit__` aún se ejecuta, antes de que la excepción se propague al ámbito externo.

Si solo necesita un método `__exit__` , puede devolver la instancia del administrador de contexto:

```
class MyContextManager:
    def __enter__(self):
        return self

    def __exit__(self):
        print('something')
```

Escribiendo tu propio administrador de contexto usando la sintaxis del generador.

También es posible escribir un administrador de contexto usando la sintaxis del generador gracias al decorador `contextlib.contextmanager` :

```
import contextlib

@contextlib.contextmanager
def context_manager(num):
    print('Enter')
    yield num + 1
    print('Exit')

with context_manager(2) as cm:
    # the following instructions are run when the 'yield' point of the context
    # manager is reached.
    # 'cm' will have the value that was yielded
    print('Right in the middle with cm = {}'.format(cm))
```

produce:

```
Enter
Right in the middle with cm = 3
Exit
```

El decorador simplifica la tarea de escribir un administrador de contexto al convertir un generador en uno. Todo antes de que la expresión de rendimiento se convierta en el método `__enter__` , el valor generado se convierta en el valor devuelto por el generador (que se puede vincular a una variable en la declaración with), y todo después de la expresión de rendimiento se convierte en el método `__exit__`

Si el administrador de contexto debe manejar una excepción, se puede escribir un `try..except..finally` try..except..finally en el generador, y este bloque de excepción manejará cualquier excepción que se genere en el bloque `with` .

```
@contextlib.contextmanager
def error_handling_context_manager(num):
    print("Enter")
    try:
        yield num + 1
    except ZeroDivisionError:
        print("Caught error")
    finally:
        print("Cleaning up")
    print("Exit")

with error_handling_context_manager(-1) as cm:
    print("Dividing by cm = {}".format(cm))
    print(2 / cm)
```

Esto produce:

```
Enter
Dividing by cm = 0
Caught error
Cleaning up
Exit
```

Gestores de contexto multiples

Puede abrir varios gestores de contenido al mismo tiempo:

```
with open(input_path) as input_file, open(output_path, 'w') as output_file:

    # do something with both files.

    # e.g. copy the contents of input_file into output_file
    for line in input_file:
        output_file.write(line + '\n')
```

Tiene el mismo efecto que los administradores de contexto de anidamiento:

```
with open(input_path) as input_file:
    with open(output_path, 'w') as output_file:
        for line in input_file:
            output_file.write(line + '\n')
```

Gestionar recursos

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):
        self.open_file.close()
```

`__init__()` configura el objeto, en este caso, configura el nombre del archivo y el modo para abrir el archivo. `__enter__()` abre y devuelve el archivo y `__exit__()` simplemente lo cierra.

El uso de estos métodos mágicos (`__enter__` , `__exit__`) le permite implementar objetos que se pueden usar fácilmente `with` la instrucción with.

Usar clase de archivo:

```
for _ in range(10000):
    with File('foo.txt', 'w') as f:
        f.write('foo')
```

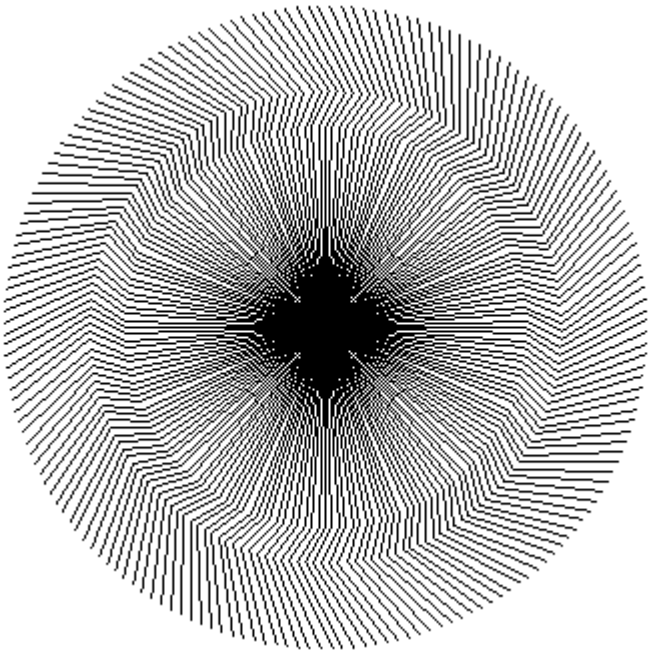
Lea Gestores de contexto (declaración “con”) en línea:

https://riptutorial.com/es/python/topic/928/gestores-de-contexto--declaracion--con--

Capítulo 92: Gráficos de tortuga

Examples

Ninja Twist (Tortuga Gráficos)



Aquí una Tortuga Gráficos Ninja Twist:

```
import turtle

ninja = turtle.Turtle()

ninja.speed(10)

for i in range(180):
    ninja.forward(100)
    ninja.right(30)
    ninja.forward(20)
    ninja.left(60)
    ninja.forward(50)
    ninja.right(30)

    ninja.penup()
    ninja.setposition(0, 0)
    ninja.pendown()

    ninja.right(2)

turtle.done()
```

Lea Gráficos de tortuga en línea: https://riptutorial.com/es/python/topic/7915/graficos-de-tortuga

Capítulo 93: hashlib

Introducción

hashlib implementa una interfaz común para muchos algoritmos de resumen de hash y resumen de mensajes. Se incluyen los algoritmos de hash seguro FIPS SHA1, SHA224, SHA256, SHA384 y SHA512.

Examples

Hash MD5 de una cadena

Este módulo implementa una interfaz común para muchos algoritmos de resumen de hash y resumen de mensajes. Se incluyen los algoritmos hash seguros SHA1, SHA224, SHA256, SHA384 y SHA512 de FIPS (definidos en FIPS 180-2), así como el algoritmo MD5 de RSA (definido en Internet RFC 1321).

Hay un método constructor nombrado para cada tipo de hash. Todos devuelven un objeto hash con la misma interfaz simple. Por ejemplo: use `sha1()` para crear un objeto hash SHA1.

```
hash.sha1()
```

Los constructores de algoritmos hash que siempre están presentes en este módulo son `md5()` , `sha1()` , `sha224()` , `sha256()` , `sha384()` y `sha512()` .

Ahora puede alimentar este objeto con cadenas arbitrarias usando el método `update()` . En cualquier momento, puede solicitar el resumen de la concatenación de las cadenas que se le han suministrado utilizando los métodos `digest()` o `hexdigest()` .

```
hash.update(arg)
```

> Actualice el objeto hash con la cadena arg. Las llamadas repetidas son equivalentes a una sola llamada con la concatenación de todos los argumentos: m.update (a); m.update (b) es equivalente a m.update (a + b).

```
hash.digest()
```

> Devuelva el resumen de las cadenas pasadas al método update () hasta el momento. Esta es una cadena de bytes digest_size que pueden contener caracteres no ASCII, incluidos los bytes nulos.

```
hash.hexdigest()
```

> Al igual que digest () excepto que el resumen se devuelve como una cadena de doble longitud, que contiene solo dígitos hexadecimales. Esto se puede usar para

intercambiar el valor de forma segura en el correo electrónico u otros entornos no binarios.

Aquí hay un ejemplo:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.hexdigest()
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
>>> m.digest_size
16
>>> m.block_size
64
```

o:

```
hashlib.md5("Nobody inspects the spammish repetition").hexdigest()
    'bb649c83dd1ea5c9d9dec9a18df0ffe9'
```

algoritmo proporcionado por OpenSSL

También existe un constructor `new()` genérico que toma el nombre de la cadena del algoritmo deseado como su primer parámetro para permitir el acceso a los hashes enumerados anteriormente, así como a cualquier otro algoritmo que pueda ofrecer su biblioteca OpenSSL. Los constructores nombrados son mucho más rápidos que `new()` y deberían preferirse.

Usando `new()` con un algoritmo proporcionado por OpenSSL:

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Lea hashlib en línea: https://riptutorial.com/es/python/topic/8980/hashlib

Capítulo 94: Heapq

Examples

Artículos más grandes y más pequeños en una colección.

Para encontrar los elementos más grandes en una colección, el módulo `heapq` tiene una función llamada `nlargest` , le pasamos dos argumentos, el primero es el número de elementos que queremos recuperar, el segundo es el nombre de la colección:

```
import heapq


numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers))  # [200, 150, 100, 50]
```

De manera similar, para encontrar los elementos más pequeños en una colección, usamos la función `nsmallest` :

```
print(heapq.nsmallest(4, numbers))  # [1, 2, 4, 8]
```

Las funciones `nlargest` y `nsmallest` toman un argumento opcional (parámetro clave) para estructuras de datos complicadas. El siguiente ejemplo muestra el uso de la propiedad `age` para recuperar las personas más antiguas y más jóvenes del diccionario de `people` :

```
people = [
    {'firstname': 'John', 'lastname': 'Doe', 'age': 30},
    {'firstname': 'Jane', 'lastname': 'Doe', 'age': 25},
    {'firstname': 'Janie', 'lastname': 'Doe', 'age': 10},
    {'firstname': 'Jane', 'lastname': 'Roe', 'age': 22},
    {'firstname': 'Johnny', 'lastname': 'Doe', 'age': 12},
    {'firstname': 'John', 'lastname': 'Roe', 'age': 45}
]

oldest = heapq.nlargest(2, people, key=lambda s: s['age'])
print(oldest)
# Output: [{'firstname': 'John', 'age': 45, 'lastname': 'Roe'}, {'firstname': 'John', 'age':
30, 'lastname': 'Doe'}]

youngest = heapq.nsmallest(2, people, key=lambda s: s['age'])
print(youngest)
# Output: [{'firstname': 'Janie', 'age': 10, 'lastname': 'Doe'}, {'firstname': 'Johnny',
'age': 12, 'lastname': 'Doe'}]
```

Artículo más pequeño en una colección.

La propiedad más interesante de un `heap` es que su elemento más pequeño es siempre el primer elemento: `heap[0]`

```
import heapq


numbers = [10, 4, 2, 100, 20, 50, 32, 200, 150, 8]

heapq.heapify(numbers)
print(numbers)
# Output: [2, 4, 10, 100, 8, 50, 32, 200, 150, 20]

heapq.heappop(numbers)  # 2
print(numbers)
# Output: [4, 8, 10, 100, 20, 50, 32, 200, 150]

heapq.heappop(numbers)  # 4
print(numbers)
# Output:  [8, 20, 10, 100, 150, 50, 32, 200]
```

Lea Heapq en línea: https://riptutorial.com/es/python/topic/7489/heapq

Capítulo 95: Herramienta 2to3

Sintaxis

- $ 2to3 [-options] ruta / a / archivo.py

Parámetros

Parámetro	Descripción
nombre de archivo / nombre_directorio	2to3 acepta una lista de archivos o directorios que se va a transformar como su argumento. Los directorios se recursivamente recorren para las fuentes de Python.
Opción	**Opción Descripción**
-f FIX, --fix = FIX	Especificar transformaciones a aplicar; por defecto: todos. Listar las transformaciones disponibles con `--list-fixes`
-j PROCESOS, --procesos = PROCESOS	Ejecutar 2to3 al mismo tiempo
-x NOFIX, --nofix = NOFIX	Excluir una transformación
-l, --list-fixes	Listar las transformaciones disponibles
-p, --print-function	Cambia la gramática para que `print()` sea considerada una función.
-v, --verbose	Salida más detallada
--no-diffs	No se muestran datos de la refactorización.
-w	Escribir archivos modificados
-n, --nobackups	No cree copias de seguridad de archivos modificados
-o OUTPUT_DIR, --output-dir = OUTPUT_DIR	Coloque los archivos de salida en este directorio en lugar de sobrescribir los archivos de entrada. Requiere la `-n` , ya que los archivos de copia de seguridad no son necesarios cuando los archivos de entrada no se modifican.
-W, --write-unchanged-files	Escribir archivos de salida, incluso si no se requieren cambios. Útil con `-o` para que se traduzca y copie un árbol fuente completo. Implica `-w` .

Parámetro	Descripción
--add-suffix = ADD_SUFFIX	Especifique una cadena que se agregará a todos los nombres de archivo de salida. Requiere `-n` si no está vacío. Ej .: `--add-suffix='3'` generará archivos `.py3` .

Observaciones

La herramienta 2to3 es un programa de Python que se utiliza para convertir el código escrito en Python 2.x en el código de Python 3.x. La herramienta lee el código fuente de Python 2.x y aplica una serie de reparadores para transformarlo en un código válido de Python 3.x.

La herramienta 2to3 está disponible en la biblioteca estándar como lib2to3, que contiene un amplio conjunto de fijadores que manejarán casi todo el código. Dado que lib2to3 es una biblioteca genérica, es posible escribir sus propios fijadores para 2to3.

Examples

Uso básico

Considere el siguiente código Python2.x. Guarde el archivo como `example.py`

Python 2.x 2.0

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

En el archivo anterior, hay varias líneas incompatibles. El método `raw_input()` se ha reemplazado con `input()` en Python 3.xy la `print` ya no es una declaración, sino una función. Este código se puede convertir a código Python 3.x usando la herramienta 2to3.

Unix

```
$ 2to3 example.py
```

Windows

```
> path/to/2to3.py example.py
```

La ejecución del código anterior generará las diferencias con respecto al archivo fuente original, como se muestra a continuación.

```
RefactoringTool: Skipping implicit fixer: buffer
```

```
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored example.py
--- example.py    (original)
+++ example.py    (refactored)
@@ -1,5 +1,5 @@
 def greet(name):
-    print "Hello, {0}!".format(name)
-print "What's your name?"
-name = raw_input()
+    print("Hello, {0}!".format(name))
+print("What's your name?")
+name = input()
 greet(name)
RefactoringTool: Files that need to be modified:
RefactoringTool: example.py
```

Las modificaciones se pueden volver a escribir en el archivo de origen utilizando el indicador -w. Se crea una copia de seguridad del archivo original llamado `example.py.bak` , a menos que se indique el distintivo -n.

Unix

```
$ 2to3 -w example.py
```

Windows

```
> path/to/2to3.py -w example.py
```

Ahora el archivo `example.py` se ha convertido de Python 2.x al código de Python 3.x.

Una vez finalizado, `example.py` contendrá el siguiente código válido de Python3.x:

Python 3.x 3.0

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

Lea Herramienta 2to3 en línea: https://riptutorial.com/es/python/topic/5320/herramienta-2to3

Capítulo 96: herramienta grafica

Introducción

Las herramientas de python se pueden usar para generar graficos.

Examples

PyDotPlus

PyDotPlus es una versión mejorada del antiguo proyecto pydot que proporciona una interfaz Python al lenguaje Dot de Graphviz.

Instalación

Para la última versión estable:

```
pip install pydotplus
```

Para la versión de desarrollo:

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

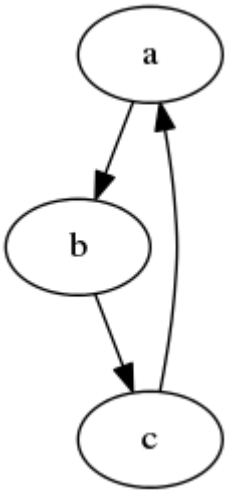
Cargar gráfico como se define por un archivo DOT

- Se asume que el archivo está en formato DOT. Se cargará, se analizará y se devolverá una clase Dot, que representa el gráfico. Por ejemplo, un simple demo.dot:

  digraph demo1 {a -> b -> c; c -> a; }

```
import pydotplus
graph_a = pydotplus.graph_from_dot_file('demo.dot')
graph_a.write_svg('test.svg') # generate graph in svg.
```

Obtendrá un svg (gráficos vectoriales escalables) como este:



PyGraphviz

Obtenga PyGraphviz del Índice de Paquetes de Python en http://pypi.python.org/pypi/pygraphviz

O instálalo con:

```
pip install pygraphviz
```

y se intentará encontrar e instalar una versión adecuada que coincida con su sistema operativo y la versión de Python.

Puede instalar la versión de desarrollo (en github.com) con:

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

Obtenga PyGraphviz del Índice de Paquetes de Python en http://pypi.python.org/pypi/pygraphviz

O instálalo con:

```
easy_install pygraphviz
```

y se intentará encontrar e instalar una versión adecuada que coincida con su sistema operativo y la versión de Python.

Cargar gráfico como se define por un archivo DOT

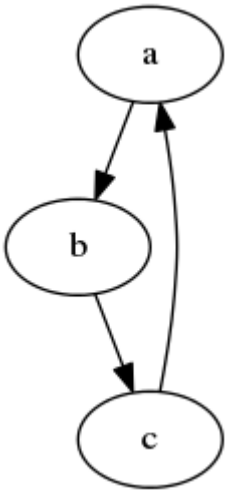
- Se asume que el archivo está en formato DOT. Se cargará, se analizará y se devolverá una clase Dot, que representa el gráfico. Por ejemplo, un simple demo.dot:

  digraph demo1 {a -> b -> c; c -> a; }

- Cárgalo y dibújalo.

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
G.draw('test', format='svg', prog='dot')
```

Obtendrá un svg (gráficos vectoriales escalables) como este:



Lea herramienta grafica en línea: https://riptutorial.com/es/python/topic/9483/herramienta-grafica

Capítulo 97: ijson

Introducción

ijson es una excelente biblioteca para trabajar con archivos JSON en Python. Desafortunadamente, de forma predeterminada, utiliza un analizador Python JSON puro como su backend. Se puede lograr un rendimiento mucho mayor utilizando un backend C.

Examples

Ejemplo simple

Ejemplo de ejemplo Tomado de un punto de referencia

```
import ijson

def load_json(filename):
    with open(filename, 'r') as fd:
        parser = ijson.parse(fd)
        ret = {'builders': {}}
        for prefix, event, value in parser:
            if (prefix, event) == ('builders', 'map_key'):
                buildername = value
                ret['builders'][buildername] = {}
            elif prefix.endswith('.shortname'):
                ret['builders'][buildername]['shortname'] = value

        return ret

if __name__ == "__main__":
    load_json('allthethings.json')
```

ENLACE DE ARCHIVO JSON

Lea ijson en línea: https://riptutorial.com/es/python/topic/8342/ijson

Capítulo 98: Implementaciones no oficiales de Python

Examples

IronPython

Implementación de código abierto para .NET y Mono escrito en C #, con licencia de Apache License 2.0. Se basa en DLR (Dynamic Language Runtime). Sólo es compatible con la versión 2.7, la versión 3 se está desarrollando actualmente.

Diferencias con CPython:

- Integración estrecha con .NET Framework.
- Las cadenas son Unicode por defecto.
- No admite extensiones para CPython escritas en C.
- No sufre de Global Intérprete Lock.
- El rendimiento suele ser menor, aunque depende de las pruebas.

Hola Mundo

```
print "Hello World!"
```

También puedes usar las funciones .NET:

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

enlaces externos

- Sitio web oficial
- Repositorio GitHub

Jython

Implementación de código abierto para JVM escrita en Java, con licencia de Python Software Foundation License. Sólo es compatible con la versión 2.7, la versión 3 se está desarrollando actualmente.

Diferencias con CPython:

- Integración estrecha con JVM.
- Las cuerdas son Unicode.
- No admite extensiones para CPython escritas en C.
- No sufre de Global Intérprete Lock.
- El rendimiento suele ser menor, aunque depende de las pruebas.

Hola Mundo

```
print "Hello World!"
```

También puedes usar las funciones de Java:

```
from java.lang import System
System.out.println("Hello World!")
```

enlaces externos

- Sitio web oficial
- Repositorio mercurial

Transcrypt

Transcrypt es una herramienta para precompilar un subconjunto bastante extenso de Python en un Javascript compacto y legible. Tiene las siguientes características:

- Permite la programación OO clásica con herencia múltiple utilizando la sintaxis de Python pura, analizada por el analizador nativo de CPython
- Integración perfecta con el universo de bibliotecas de JavaScript de alta calidad orientadas a la web, en lugar de las de Python orientadas al escritorio
- Sistema de módulo jerárquico basado en URL que permite la distribución de módulos a través de PyPi
- Relación simple entre la fuente Python y el código JavaScript generado para una fácil depuración
- Mapas de referencia de niveles múltiples y anotación opcional del código de destino con referencias de origen
- Descargas compactas, kB's en lugar de MB's
- Código JavaScript optimizado, que utiliza memoization (almacenamiento en caché de llamadas) para omitir opcionalmente la cadena de búsqueda de prototipos
- La sobrecarga del operador se puede activar y desactivar localmente para facilitar las matemáticas numéricas legibles

Tamaño y velocidad del código

La experiencia ha demostrado que 650 kB de código fuente de Python se traducen aproximadamente en la misma cantidad de código fuente de JavaScript. La velocidad coincide con la velocidad del JavaScript manuscrito y puede sobrepasarlo si la memorización de llamadas está activada.

Integración con HTML

```
<script src="__javascript__/hello.js"></script>
<h2>Hello demo</h2>

<p>
<div id = "greet">...</div>
<button onclick="hello.solarSystem.greet ()">Click me repeatedly!</button>

<p>
<div id = "explain">...</div>
<button onclick="hello.solarSystem.explain ()">And click me repeatedly too!</button>
```

Integración con JavaScript y DOM

```
from itertools import chain

class SolarSystem:
    planets = [list (chain (planet, (index + 1,))) for index, planet in enumerate ((
        ('Mercury', 'hot', 2240),
        ('Venus', 'sulphurous', 6052),
        ('Earth', 'fertile', 6378),
        ('Mars', 'reddish', 3397),
        ('Jupiter', 'stormy', 71492),
        ('Saturn', 'ringed', 60268),
        ('Uranus', 'cold', 25559),
        ('Neptune', 'very cold', 24766)
    ))]

    lines = (
        '{} is a {} planet',
        'The radius of {} is {} km',
        '{} is planet nr. {} counting from the sun'
    )

    def __init__ (self):
        self.lineIndex = 0

    def greet (self):
        self.planet = self.planets [int (Math.random () * len (self.planets))]
        document.getElementById ('greet') .innerHTML = 'Hello {}'.format (self.planet [0])
        self.explain ()

    def explain (self):
        document.getElementById ('explain').innerHTML = (
            self.lines [self.lineIndex] .format (self.planet [0], self.planet [self.lineIndex
+ 1])
        )
        self.lineIndex = (self.lineIndex + 1) % 3
```

```
        solarSystem = SolarSystem ()
```

Integración con otras bibliotecas de JavaScript

Transcrypt se puede utilizar en combinación con cualquier biblioteca de JavaScript sin medidas ni sintaxis especiales. En la documentación se dan ejemplos para ao react.js, riot.js, fabric.js y node.js.

Relación entre Python y código JavaScript

Pitón

```
class A:
    def __init__ (self, x):
        self.x = x

    def show (self, label):
        print ('A.show', label, self.x)

class B:
    def __init__ (self, y):
        alert ('In B constructor')
        self.y = y

    def show (self, label):
        print ('B.show', label, self.y)

class C (A, B):
    def __init__ (self, x, y):
        alert ('In C constructor')
        A.__init__ (self, x)
        B.__init__ (self, y)
        self.show ('constructor')

    def show (self, label):
        B.show (self, label)
        print ('C.show', label, self.x, self.y)

a = A (1001)
a.show ('america')

b = B (2002)
b.show ('russia')

c = C (3003, 4004)
c.show ('netherlands')

show2 = c.show
show2 ('copy')
```

JavaScript

```
var A = __class__ ('A', [object], {
    get __init__ () {return __get__ (this, function (self, x) {
        self.x = x;
    });},
    get show () {return __get__ (this, function (self, label) {
        print ('A.show', label, self.x);
    });}
});
var B = __class__ ('B', [object], {
    get __init__ () {return __get__ (this, function (self, y) {
        alert ('In B constructor');
        self.y = y;
    });},
    get show () {return __get__ (this, function (self, label) {
        print ('B.show', label, self.y);
    });}
});
var C = __class__ ('C', [A, B], {
    get __init__ () {return __get__ (this, function (self, x, y) {
        alert ('In C constructor');
        A.__init__ (self, x);
        B.__init__ (self, y);
        self.show ('constructor');
    });},
    get show () {return __get__ (this, function (self, label) {
        B.show (self, label);
        print ('C.show', label, self.x, self.y);
    });}
});
var a = A (1001);
a.show ('america');
var b = B (2002);
b.show ('russia');
var c = C (3003, 4004);
c.show ('netherlands');
var show2 = c.show;
show2 ('copy');
```

enlaces externos

- Sitio web oficial: http://www.transcrypt.org/
- Repositorio: https://github.com/JdeH/Transcrypt

Lea Implementaciones no oficiales de Python en línea:
https://riptutorial.com/es/python/topic/5225/implementaciones-no-oficiales-de-python

Capítulo 99: Importando modulos

Sintaxis

- importar *nombre_módulo*
- import *module_name.submodule_name*
- desde *nombre_módulo* import *
- from *module_name* import *submodule_name* [, *class_name* , *function_name* , ... etc]
- desde *nombre_módulo* importar *nombre_algunos* como *nuevo_nombre*
- from *module_name.submodule_name* import *class_name* [, *function_name* , ... etc]

Observaciones

Importar un módulo hará que Python evalúe todo el código de nivel superior en este módulo para que *aprenda* todas las funciones, clases y variables que contiene el módulo. Cuando desee que un módulo suyo se importe en otro lugar, tenga cuidado con su código de nivel superior y `if __name__ == '__main__':` en `if __name__ == '__main__':` si no desea que se ejecute cuando se importe el módulo.

Examples

Importando un modulo

Utilice la declaración de `import` :

```
>>> import random
>>> print(random.randint(1, 10))
4
```

`import module` importará un módulo y luego le permitirá hacer referencia a sus objetos (valores, funciones y clases, por ejemplo) utilizando la sintaxis `module.name` . En el ejemplo anterior, se importa el módulo `random` , que contiene la función `randint` . Entonces, al importar `random` , puede llamar a `randint` con `random.randint` .

Puedes importar un módulo y asignarlo a un nombre diferente:

```
>>> import random as rn
>>> print(rn.randint(1, 10))
4
```

Si su archivo python `main.py` está en la misma carpeta que `custom.py` . Puedes importarlo así:

```
import custom
```

También es posible importar una función desde un módulo:

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

Para importar funciones específicas más profundamente en un módulo, el operador de punto se puede usar **solo** en el lado izquierdo de la palabra clave de `import` :

```
from urllib.request import urlopen
```

En Python, tenemos dos formas de llamar a la función desde el nivel superior. Uno es `import` y otro es `from` . Debemos utilizar la `import` cuando tenemos la posibilidad de colisión de nombres. Supongamos que tenemos el archivo `hello.py` y los archivos `world.py` que tienen la misma función llamada `function` . Entonces la declaración de `import` funcionará bien.

```
from hello import function
from world import function

function() #world's function will be invoked. Not hello's
```

En general la `import` le proporcionará un espacio de nombres.

```
import hello
import world

hello.function() # exclusively hello's function will be invoked
world.function() # exclusively world's function will be invoked
```

Pero si está lo suficientemente seguro, en todo su proyecto no hay forma de tener el mismo nombre de función que debe usar `from` declaración

Múltiples importaciones se pueden hacer en la misma línea:

```
>>> # Multiple modules
>>> import time, sockets, random
>>> # Multiple functions
>>> from math import sin, cos, tan
>>> # Multiple constants
>>> from math import pi, e

>>> print(pi)
3.141592653589793
>>> print(cos(45))
0.5253219888177297
>>> print(time.time())
1482807222.7240417
```

Las palabras clave y la sintaxis que se muestran arriba también se pueden usar en combinaciones:

```
>>> from urllib.request import urlopen as geturl, pathname2url as path2url, getproxies
>>> from math import factorial as fact, gamma, atan as arctan
>>> import random.randint, time, sys
```

```
>>> print(time.time())
1482807222.7240417
>>> print(arctan(60))
1.554131203080956
>>> filepath = "/dogs/jumping poodle (december).png"
>>> print(path2url(filepath))
/dogs/jumping%20poodle%20%28december%29.png
```

Importando nombres específicos desde un módulo

En lugar de importar el módulo completo, solo puede importar nombres específicos:

```
from random import randint # Syntax "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))      # Out: 5
```

se necesita de forma `from random` , porque el intérprete de python debe saber de qué recurso debe importar una función o clase y `import randint` especifica la función o la clase en sí.

Otro ejemplo a continuación (similar al anterior):

```
from math import pi
print(pi)                  # Out: 3.14159265359
```

El siguiente ejemplo generará un error, porque no hemos importado un módulo:

```
random.randrange(1, 10)    # works only if "import random" has been run before
```

Salidas:

```
NameError: name 'random' is not defined
```

El intérprete de python no entiende lo que quieres decir con `random` . Debe declararse agregando `import random` al ejemplo:

```
import random
random.randrange(1, 10)
```

Importando todos los nombres de un módulo

```
from module_name import *
```

por ejemplo:

```
from math import *
sqrt(2)    # instead of math.sqrt(2)
ceil(2.7)  # instead of math.ceil(2.7)
```

Esto importará todos los nombres definidos en el módulo `math` en el espacio de nombres global, excepto los nombres que comienzan con un guión bajo (lo que indica que el escritor siente que es solo para uso interno).

Advertencia : si una función con el mismo nombre ya se definió o importó, se **sobrescribirá** . Casi siempre importando solo nombres específicos `from math import sqrt, ceil` es la **forma recomendada** :

```
def sqrt(num):
    print("I don't know what's the square root of {}.".format(num))

sqrt(4)
# Output: I don't know what's the square root of 4.

from math import *
sqrt(4)
# Output: 2.0
```

Las importaciones destacadas solo se permiten a nivel de módulo. Los intentos de realizarlas en definiciones de clase o función dan como resultado un `SyntaxError` .

```
def f():
    from math import *
```

y

```
class A:
    from math import *
```

ambos fallan con:

```
SyntaxError: import * only allowed at module level
```

La variable especial __all__

Los módulos pueden tener una variable especial llamada `__all__` para restringir qué variables se importan cuando se usan `from mymodule import *` .

Dado el siguiente módulo:

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

Sólo `imported_by_star` ha sido importada cuando se utiliza `from mymodule import *` :

```
>>> from mymodule import *
```

```
>>> imported_by_star
42
>>> not_imported_by_star
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'not_imported_by_star' is not defined
```

Sin embargo, `not_imported_by_star` se puede importar explícitamente:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
21
```

Importación programática

Python 2.x 2.7

Para importar un módulo a través de una llamada de función, use el módulo `importlib` (incluido en Python a partir de la versión 2.7):

```
import importlib
random = importlib.import_module("random")
```

La función `importlib.import_module()` también importará el submódulo de un paquete directamente:

```
collections_abc = importlib.import_module("collections.abc")
```

Para versiones anteriores de Python, use el módulo `imp` .

Python 2.x 2.7

Utilice las funciones `imp.find_module` y `imp.load_module` para realizar una importación programática.

Tomado de la documentación de la biblioteca estándar

```
import imp, sys
def import_module(name):
    fp, pathname, description = imp.find_module(name)
    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        if fp:
            fp.close()
```

NO use `__import__()` para importar módulos mediante programación. Hay detalles sutiles relacionados con `sys.modules` , el argumento `fromlist` , etc. que son fáciles de pasar por alto, que `importlib.import_module()` maneja por ti.

Importar módulos desde una ubicación de sistema de archivos arbitraria.

Si desea importar un módulo que ya no existe como un módulo integrado en la Biblioteca estándar de Python ni como un paquete adicional, puede hacerlo agregando la ruta al directorio donde se encuentra su módulo a `sys.path` Esto puede ser útil cuando existen varios entornos de Python en un host.

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

Es importante que agregue la ruta al *directorio* en el que se encuentra `mymodule` , no la ruta al módulo en sí.

Reglas PEP8 para Importaciones

Algunas pautas de estilo PEP8 recomendadas para importaciones:

1. Las importaciones deben ser en líneas separadas:

   ```
   from math import sqrt, ceil      # Not recommended
   from math import sqrt            # Recommended
   from math import ceil
   ```

2. Ordene las importaciones de la siguiente manera en la parte superior del módulo:

   - Importaciones de la biblioteca estándar
   - Importaciones de terceros relacionados
   - Importaciones específicas de aplicaciones / bibliotecas locales

3. Se deben evitar las importaciones de comodines ya que genera confusión en los nombres en el espacio de nombres actual. Si lo hace `from module import *` , puede no estar claro si un nombre específico en su código proviene de `module` o no. Esto es doblemente cierto si tiene múltiples declaraciones `from module import *` -type.

4. Evite el uso de importaciones relativas; usa importaciones explícitas en su lugar.

Importando submódulos

```
from module.submodule import function
```

Esto importa la `function` desde `module.submodule` .

__importar __ () función

La función `__import__()` se puede usar para importar módulos donde el nombre solo se conoce en tiempo de ejecución

```
if user_input == "os":
    os = __import__("os")
```

```
# equivalent to import os
```

Esta función también se puede utilizar para especificar la ruta del archivo a un módulo

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

Reimportando un módulo

Cuando utilice el intérprete interactivo, es posible que desee volver a cargar un módulo. Esto puede ser útil si está editando un módulo y desea importar la versión más reciente, o si ha modificado un elemento de un módulo existente y desea revertir sus cambios.

Tenga en cuenta que no **puede** volver a `import` el módulo para revertir:

```
import math
math.pi = 3
print(math.pi)    # 3
import math
print(math.pi)    # 3
```

Esto se debe a que el intérprete registra todos los módulos que importa. Y cuando intenta reimportar un módulo, el intérprete lo ve en el registro y no hace nada. Por lo tanto, la manera más difícil de reimportar es usar la `import` después de eliminar el elemento correspondiente del registro:

```
print(math.pi)    # 3
import sys
if 'math' in sys.modules:  # Is the ``math`` module in the register?
    del sys.modules['math']  # If so, remove it.
import math
print(math.pi)    # 3.141592653589793
```

Pero hay más de una manera directa y sencilla.

Python 2

Utilice la función de `reload` :

Python 2.x 2.3

```
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

Python 3

La función de `reload` ha movido a `importlib` :

Python 3.x 3.0

```
import math
math.pi = 3
print(math.pi)    # 3
from importlib import reload
reload(math)
print(math.pi)    # 3.141592653589793
```

Lea Importando modulos en línea: https://riptutorial.com/es/python/topic/249/importando-modulos

Capítulo 100: Incompatibilidades que se mueven de Python 2 a Python 3

Introducción

A diferencia de la mayoría de los idiomas, Python admite dos versiones principales. Desde 2008, cuando se lanzó Python 3, muchos han hecho la transición, mientras que muchos no lo han hecho. Para entender ambos, esta sección cubre las diferencias importantes entre Python 2 y Python 3.

Observaciones

Actualmente hay dos versiones compatibles de Python: 2.7 (Python 2) y 3.6 (Python 3). Además, las versiones 3.3 y 3.4 reciben actualizaciones de seguridad en formato de origen.

Python 2.7 es compatible con versiones anteriores de la mayoría de las versiones anteriores de Python, y puede ejecutar el código de Python desde la mayoría de las versiones 1.xy 2.x de Python sin cambios. Está ampliamente disponible, con una extensa colección de paquetes. También es considerado obsoleto por los desarrolladores de CPython, y recibe solo seguridad y desarrollo de corrección de errores. Los desarrolladores de CPython tienen la intención de abandonar esta versión del lenguaje en 2020 .

De acuerdo con la Propuesta 373 de mejora de Python, no hay lanzamientos futuros planeados de Python 2 después del 25 de junio de 2016, pero las correcciones de errores y las actualizaciones de seguridad serán compatibles hasta 2020. (No especifica cuál será la fecha exacta en 2020 para la fecha de caducidad de Python 2.)

Python 3 rompió intencionalmente la compatibilidad con versiones anteriores, para abordar las preocupaciones que los desarrolladores de idiomas tenían con el núcleo del lenguaje. Python 3 recibe nuevos desarrollos y nuevas características. Es la versión del lenguaje con la que los desarrolladores del lenguaje pretenden avanzar.

Durante el tiempo entre la versión inicial de Python 3.0 y la versión actual, algunas características de Python 3 se portaron en Python 2.6, y otras partes de Python 3 se ampliaron para tener una sintaxis compatible con Python 2. Por lo tanto, es posible escribir Python que funcionará tanto en Python 2 como en Python 3, mediante el uso de futuras importaciones y módulos especiales (como **seis**).

Las importaciones futuras deben estar al comienzo de su módulo:

```
from __future__ import print_function
# other imports and instructions go after __future__
print('Hello world')
```

Para obtener más información sobre el módulo `__future__` , consulte la página correspondiente en

la documentación de Python .

La herramienta 2to3 es un programa de Python que convierte el código de Python 2.x en el código de Python 3.x; consulte también la documentación de Python .

El paquete seis proporciona utilidades para la compatibilidad con Python 2/3:

- Acceso unificado a bibliotecas renombradas
- variables para los tipos de cadena / Unicode
- Funciones para el método que se eliminó o se ha renombrado.

Una referencia para las diferencias entre Python 2 y Python 3 se puede encontrar aquí .

Examples

Declaración de impresión vs. función de impresión

En Python 2, `print` es una declaración:

Python 2.x 2.7

```
print "Hello World"
print                           # print a newline
print "No newline",             # add trailing comma to remove newline
print >>sys.stderr, "Error"     # print to stderr
print("hello")                  # print "hello", since ("hello") == "hello"
print()                         # print an empty tuple "()"
print 1, 2, 3                   # print space-separated arguments: "1 2 3"
print(1, 2, 3)                  # print tuple "(1, 2, 3)"
```

En Python 3, `print()` es una función, con argumentos de palabras clave para usos comunes:

Python 3.x 3.0

```
print "Hello World"              # SyntaxError
print("Hello World")
print()                         # print a newline (must use parentheses)
print("No newline", end="")      # end specifies what to append (defaults to newline)
print("Error", file=sys.stderr)  # file specifies the output buffer
print("Comma", "separated", "output", sep=",")  # sep specifies the separator
print("A", "B", "C", sep="")     # null string for sep: prints as ABC
print("Flush this", flush=True)  # flush the output buffer, added in Python 3.3
print(1, 2, 3)                   # print space-separated arguments: "1 2 3"
print((1, 2, 3))                 # print tuple "(1, 2, 3)"
```

La función de impresión tiene los siguientes parámetros:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep` es lo que separa los objetos que pasas para imprimir. Por ejemplo:

```
print('foo', 'bar', sep='~') # out: foo~bar
print('foo', 'bar', sep='.') # out: foo.bar
```

`end` es lo que sigue el final de la declaración de impresión. Por ejemplo:

```
print('foo', 'bar', end='!') # out: foo bar!
```

La impresión de nuevo después de una declaración de impresión final que no sea de nueva línea *se* imprimirá en la misma línea:

```
print('foo', end='~')
print('bar')
# out: foo~bar
```

Nota: Para compatibilidad futura, `print` *función de* `print` también está disponible en Python 2.6 en adelante; sin embargo, no se puede utilizar a menos que el análisis de la *declaración de* `print` se deshabilite con

```
from __future__ import print_function
```

Esta función tiene exactamente el mismo formato que Python 3, excepto que carece del parámetro de `flush` .

Ver PEP 3105 para la justificación.

Cuerdas: Bytes contra Unicode

Python 2.x 2.7

En Python 2 hay dos variantes de cadena: las de bytes con tipo (`str`) y las de texto con tipo (`unicode`).

En Python 2, un objeto de tipo `str` es siempre una secuencia de bytes, pero se usa comúnmente tanto para texto como para datos binarios.

Un literal de cadena se interpreta como una cadena de bytes.

```
s = 'Cafe'    # type(s) == str
```

Hay dos excepciones: Puedes definir un *literal de Unicode (texto)* explícitamente prefijando el literal con `u` :

```
s = u'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == str
```

Alternativamente, puede especificar que los literales de cadena de un módulo completo deberían crear literales Unicode (texto):

```
from __future__ import unicode_literals

s = 'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == unicode
```

Para verificar si su variable es una cadena (ya sea Unicode o una cadena de bytes), puede usar:

```
isinstance(s, basestring)
```

Python 3.x 3.0

En Python 3, el tipo `str` es un tipo de texto Unicode.

```
s = 'Cafe'           # type(s) == str
s = 'Café'           # type(s) == str (note the accented trailing e)
```

Además, Python 3 agregó un objeto de `bytes` , adecuado para "blobs" binarios o escrito en archivos independientes de la codificación. Para crear un objeto de bytes, puede prefijar `b` a un literal de cadena o llamar al método de `encode` la cadena:

```
# Or, if you really need a byte string:
s = b'Cafe'          # type(s) == bytes
s = 'Café'.encode()  # type(s) == bytes
```

Para probar si un valor es una cadena, use:

```
isinstance(s, str)
```

Python 3.x 3.3

También es posible prefijar literales de cadena con un prefijo `u` para facilitar la compatibilidad entre las bases de código de Python 2 y Python 3. Dado que, en Python 3, todas las cadenas son Unicode de forma predeterminada, el antepuesto de una cadena literal con `u` no tiene ningún efecto:

```
u'Cafe' == 'Cafe'
```

Sin embargo, el prefijo `ur` cadena Unicode sin procesar de Python 2 no es compatible:

```
>>> ur'Café'
  File "<stdin>", line 1
    ur'Café'
            ^
SyntaxError: invalid syntax
```

Tenga en cuenta que debe `encode` un objeto de texto (`str`) de Python 3 para convertirlo en una representación de `bytes` de ese texto. La codificación predeterminada de este método es UTF-8 .

Puede usar `decode` para pedir a un objeto de `bytes` texto Unicode que representa:

```
>>> b.decode()
'Café'
```

Python 2.x 2.6

Mientras que el tipo de `bytes` existe tanto en Python 2 como en 3, el tipo `unicode` solo existe en Python 2. Para usar las cadenas de Unicode implícitas de Python 3 en Python 2, agregue lo siguiente en la parte superior de su archivo de código:

```
from __future__ import unicode_literals
print(repr("hi"))
# u'hi'
```

Python 3.x 3.0

Otra diferencia importante es que la indexación de bytes en Python 3 da como resultado una salida `int` como:

```
b"abc"[0] == 97
```

Mientras se corta en un tamaño de uno, se obtiene un objeto de longitud de 1 bytes:

```
b"abc"[0:1] == b"a"
```

Además, Python 3 corrige algunos comportamientos inusuales con Unicode, es decir, invierte cadenas de bytes en Python 2. Por ejemplo, se resuelve el siguiente problema :

```
# -*- coding: utf8 -*-
print("Hi, my name is Łukasz Langa.")
print(u"Hi, my name is Łukasz Langa."[::-1])
print("Hi, my name is Łukasz Langa."[::-1])

# Output in Python 2
# Hi, my name is Łukasz Langa.
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsaku◆◆ si eman ym ,iH

# Output in Python 3
# Hi, my name is Łukasz Langa.
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsakuŁ si eman ym ,iH
```

División entera

El **símbolo de división** estándar (`/`) funciona de manera diferente en Python 3 y Python 2 cuando se aplica a enteros.

Cuando se divide un entero por otro entero en Python 3, la operación de división `x / y` representa una **verdadera división** (usa el método `__truediv__`) y produce un resultado de punto flotante. Mientras tanto, la misma operación en Python 2 representa una **división clásica** que redondea el resultado hacia el infinito negativo (también conocido como tomar el *piso*).

Por ejemplo:

Código	Salida Python 2	Salida Python 3
`3 / 2`	1	1.5
`2 / 3`	0	0.6666666666666666
`-3 / 2`	-2	-1.5

El comportamiento de redondeo hacia cero fue obsoleto en Python 2.2 , pero permanece en Python 2.7 por motivos de compatibilidad con versiones anteriores y se eliminó en Python 3.

Nota: para obtener un resultado *flotante* en Python 2 (sin redondear el piso) podemos especificar uno de los operandos con el punto decimal. El ejemplo anterior de `2/3` que da `0` en Python 2 se usará como `2 / 3.0` o `2.0 / 3` o `2.0/3.0` para obtener `0.6666666666666666`

Código	Salida Python 2	Salida Python 3
`3.0 / 2.0`	1.5	1.5
`2 / 3.0`	0.6666666666666666	0.6666666666666666
`-3.0 / 2`	-1.5	-1.5

También está el **operador de división de piso** (`//`), que funciona de la misma manera en ambas versiones: se redondea al entero más cercano. (aunque se devuelve un flotador cuando se usa con flotadores) En ambas versiones, el operador `//` asigna a `__floordiv__` .

Código	Salida Python 2	Salida Python 3
`3 // 2`	1	1
`2 // 3`	0	0
`-3 // 2`	-2	-2
`3.0 // 2.0`	1.0	1.0
`2.0 // 3`	0.0	0.0
`-3 // 2.0`	-2.0	-2.0

Uno puede imponer explícitamente la división verdadera o la división de piso usando funciones nativas en el módulo del `operator` :

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25           # equivalent to `/` in Python 3
```

```
assert floordiv(10, 8) == 1             # equivalent to `//`
```

Si bien es claro y explícito, el uso de funciones de operador para cada división puede ser tedioso. Cambiar el comportamiento del operador `/` a menudo será preferido. Una práctica común es eliminar el comportamiento de división típico agregando la `from __future__ import division` como la primera declaración en cada módulo:

```
# needs to be the first statement in a module
from __future__ import division
```

Código	Salida Python 2	Salida Python 3
`3 / 2`	1.5	1.5
`2 / 3`	0.6666666666666666	0.6666666666666666
`-3 / 2`	-1.5	-1.5

`from __future__ import division` garantiza que el operador `/` representa la división verdadera y solo dentro de los módulos que contienen la importación `__future__` , por lo que no hay razones de peso para no habilitarla en todos los módulos nuevos.

Nota : Algunos otros lenguajes de programación utilizan el *redondeo hacia cero* (truncamiento) en lugar de *redondear hacia el infinito negativo* como Python (es decir, en esos idiomas `-3 / 2 == -1`). Este comportamiento puede crear confusión al portar o comparar código.

Nota sobre los operandos de flotación : Como alternativa a la `from __future__ import division` , se podría usar el símbolo de división habitual `/` y asegurarse de que al menos uno de los operandos es una flotación: `3 / 2.0 == 1.5` . Sin embargo, esto puede considerarse una mala práctica. Es demasiado fácil escribir el `average = sum(items) / len(items)` y olvidarse de lanzar uno de los argumentos para flotar. Además, estos casos pueden evadir con frecuencia el aviso durante las pruebas, por ejemplo, si realiza pruebas en una matriz que contiene `float` pero recibe una matriz de `int` s en producción. Además, si se usa el mismo código en Python 3, los programas que esperan que 3/2 `3 / 2 == 1` sea Verdadero no funcionarán correctamente.

Vea PEP 238 para una explicación más detallada de por qué se cambió el operador de la división en Python 3 y por qué se debe evitar la división de estilo antiguo.

Vea el tema de *Matemáticas simples* para más información sobre la división.

Reducir ya no es una función incorporada.

En Python 2, `reduce` está disponible como una función incorporada o desde el paquete `functools` (versión 2.6 en adelante), mientras que en Python 3, `reduce` está disponible solo desde `functools` . Sin embargo, la sintaxis para `reduce` tanto en Python2 como en Python3 es la misma y es `reduce(function_to_reduce, list_to_reduce)` .

Como ejemplo, consideremos reducir una lista a un solo valor al dividir cada uno de los números adyacentes. Aquí usamos la función `truediv` de la biblioteca del `operator` .

En Python 2.x es tan simple como:

Python 2.x 2.3

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

En Python 3.x el ejemplo se vuelve un poco más complicado:

Python 3.x 3.0

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

También podemos utilizar `from functools import reduce` para evitar `reduce` llamadas con el nombre del espacio de nombres.

Diferencias entre las funciones de rango y rango

En Python 2, la función de `range` devuelve una lista, mientras que `xrange` crea un objeto de `xrange` especial, que es una secuencia inmutable, que a diferencia de otros tipos de secuencia incorporados, no admite el corte y no tiene métodos de `index` ni de `count` :

Python 2.x 2.3

```
print(range(1, 10))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(isinstance(range(1, 10), list))
# Out: True

print(xrange(1, 10))
# Out: xrange(1, 10)

print(isinstance(xrange(1, 10), xrange))
# Out: True
```

En Python 3, `xrange` se expandió a la secuencia de `range` , que ahora crea un objeto de `range` . No hay tipo `xrange` :

Python 3.x 3.0

```
print(range(1, 10))
# Out: range(1, 10)

print(isinstance(range(1, 10), range))
```

```
# Out: True

# print(xrange(1, 10))
# The output will be:
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#NameError: name 'xrange' is not defined
```

Además, dado que Python 3.2, el `range` también admite el corte, `index` y `count` :

```
print(range(1, 10)[3:7])
# Out: range(3, 7)
print(range(1, 10).count(5))
# Out: 1
print(range(1, 10).index(7))
# Out: 6
```

La ventaja de usar un tipo de secuencia especial en lugar de una lista es que el intérprete no tiene que asignar memoria para una lista y llenarla:

Python 2.x 2.3

```
# range(10000000000000000)
# The output would be:
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# MemoryError

print(xrange(100000000000000000))
# Out: xrange(100000000000000000)
```

Como generalmente se desea el último comportamiento, el primero se eliminó en Python 3. Si aún desea tener una lista en Python 3, simplemente puede usar el constructor `list()` en un objeto de `range` :

Python 3.x 3.0

```
print(list(range(1, 10)))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Compatibilidad

Para mantener la compatibilidad entre las versiones de Python 2.xy Python 3.x, puede usar el módulo `builtins` del `future` paquete externo para lograr tanto *compatibilidad con versiones anteriores* como *compatibilidad con versiones anteriores* :

Python 2.x 2.0

```
#forward-compatible
from builtins import range
```

```
for i in range(10**8):
    pass
```

Python 3.x 3.0

```
#backward-compatible
from past.builtins import xrange

for i in xrange(10**8):
    pass
```

El `range` en la biblioteca `future` admite la segmentación, el `index` y el `count` en todas las versiones de Python, al igual que el método incorporado en Python 3.2+.

Desembalaje Iterables

Python 3.x 3.0

En Python 3, puedes descomprimir un iterable sin saber la cantidad exacta de elementos que contiene, e incluso tener una variable que mantenga el final del iterable. Para eso, proporciona una variable que puede recopilar una lista de valores. Esto se hace colocando un asterisco antes del nombre. Por ejemplo, desempaquetar una `list` :

```
first, second, *tail, last = [1, 2, 3, 4, 5]
print(first)
# Out: 1
print(second)
# Out: 2
print(tail)
# Out: [3, 4]
print(last)
# Out: 5
```

Nota : Al usar la sintaxis de la `*variable` , la `variable` siempre será una lista, incluso si el tipo original no era una lista. Puede contener cero o más elementos dependiendo del número de elementos en la lista original.

```
first, second, *tail, last = [1, 2, 3, 4]
print(tail)
# Out: [3]

first, second, *tail, last = [1, 2, 3]
print(tail)
# Out: []
print(last)
# Out: 3
```

Del mismo modo, desempaquetando un `str` :

```
begin, *tail = "Hello"
print(begin)
# Out: 'H'
print(tail)
```

```
# Out: ['e', 'l', 'l', 'o']
```

Ejemplo de desempaquetar una `date` ; `_` se utiliza en este ejemplo como una variable desechable (solo nos interesa el valor del `year`):

```
person = ('John', 'Doe', (10, 16, 2016))
*_, (*_, year_of_birth) = person
print(year_of_birth)
# Out: 2016
```

Vale la pena mencionar que, dado que `*` consume una cantidad variable de elementos, no puede tener dos `*` s *para el mismo iterable* en una asignación; no sabría cuántos elementos entran en el primer desempaquetado y cuántos en el segundo :

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
```

Python 3.x 3.5

Hasta ahora hemos discutido el desempaquetado en las tareas. `*` y `**` se extendieron en Python 3.5 . Ahora es posible tener varias operaciones de desempaquetado en una expresión:

```
{*range(4), 4, *(5, 6, 7)}
# Out: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x 2.0

También es posible descomprimir un iterable en argumentos de función:

```
iterable = [1, 2, 3, 4, 5]
print(iterable)
# Out: [1, 2, 3, 4, 5]
print(*iterable)
# Out: 1 2 3 4 5
```

Python 3.x 3.5

Desembalar un diccionario usa dos estrellas adyacentes `**` (PEP 448):

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
 # Out: {'x': 1, 'y': 2, 'z': 3}
```

Esto permite anular valores antiguos y fusionar diccionarios.

```
dict1 = {'x': 1, 'y': 1}
dict2 = {'y': 2, 'z': 3}
{**dict1, **dict2}
# Out: {'x': 1, 'y': 2, 'z': 3}
```

Python 3.x 3.0

Python 3 eliminó el desempaquetado de tuplas en las funciones. Por lo tanto lo siguiente no funciona en Python 3

```
# Works in Python 2, but syntax error in Python 3:
map(lambda (x, y): x + y, zip(range(5), range(5)))
# Same is true for non-lambdas:
def example((x, y)):
    pass

# Works in both Python 2 and Python 3:
map(lambda x: x[0] + x[1], zip(range(5), range(5)))
# And non-lambdas, too:
def working_example(x_y):
    x, y = x_y
    pass
```

Ver PEP 3113 para una explicación detallada.

Levantando y manejando excepciones

Esta es la sintaxis de Python 2, tenga en cuenta las comas `,` en las líneas de `raise` y `except` :

Python 2.x 2.3

```
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

En Python 3, el `,` la sintaxis se deja caer y se sustituye por el paréntesis y `as` palabra clave:

```
try:
    raise IOError("input/output error")
except IOError as exc:
    print(exc)
```

Para la compatibilidad con versiones anteriores, la sintaxis de Python 3 también está disponible en Python 2.6 en adelante, por lo que debe usarse para todo el código nuevo que no necesita ser compatible con versiones anteriores.

Python 3.x 3.0

Python 3 también agrega el encadenamiento de excepciones , en el que puede indicar que alguna otra excepción fue la *causa* de esta excepción. Por ejemplo

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from e
```

La excepción generada en la declaración de `except` es de tipo `DatabaseError` , pero la excepción original está marcada como el atributo `__cause__` de esa excepción. Cuando se muestra el rastreo,

la excepción original también se mostrará en el rastreo:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Si lanza un bloque de `except` *sin* encadenamiento explícito:

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}')
```

La traza es

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Python 2.x 2.0

Ninguno de los dos está soportado en Python 2.x; la excepción original y su rastreo se perderán si se genera otra excepción en el bloque de excepción. El siguiente código puede ser usado para compatibilidad:

```
import sys
import traceback

try:
    funcWithError()
except:
    sys_vers = getattr(sys, 'version_info', (0,))
    if sys_vers < (3, 0):
        traceback.print_exc()
    raise Exception("new exception")
```

Python 3.x 3.3

Para "olvidar" la excepción lanzada anteriormente, use `raise from None`

```
try:
    file = open('database.db')
except FileNotFoundError as e:
```

```
    raise DatabaseError('Cannot open {}') from None
```

Ahora el rastreo sería simplemente

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

O para que sea compatible con Python 2 y 3 puede usar el paquete seis así:

```
import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('Cannot open {}'), None)
```

.next () método en los iteradores renombrados

En Python 2, se puede recorrer un iterador usando un método llamado `next` en el mismo iterador:

Python 2.x 2.3

```
g = (i for i in range(0, 3))
g.next()  # Yields 0
g.next()  # Yields 1
g.next()  # Yields 2
```

En Python 3, el método `.next` ha sido renombrado a `.__next__` , reconociendo su función "mágica", por lo que llamar a `.next` generará un `AttributeError` . La forma correcta de acceder a esta funcionalidad tanto en Python 2 como en Python 3 es llamar a la `next` *función* con el iterador como argumento.

Python 3.x 3.0

```
g = (i for i in range(0, 3))
next(g)  # Yields 0
next(g)  # Yields 1
next(g)  # Yields 2
```

Este código es portátil en las versiones desde 2.6 hasta las versiones actuales.

Comparación de diferentes tipos

Python 2.x 2.3

Se pueden comparar objetos de diferentes tipos. Los resultados son arbitrarios, pero consistentes. Están ordenados de modo que `None` es menor que cualquier otra cosa, los tipos numéricos son más pequeños que los tipos no numéricos y todo lo demás está ordenado lexicográficamente por tipo. Por lo tanto, un `int` es menor que un `str` y una `tuple` es mayor que una `list` :

```
[1, 2] > 'foo'
# Out: False
(1, 2) > 'foo'
# Out: True
[1, 2] > (1, 2)
# Out: False
100 < [1, 'x'] < 'xyz' < (1, 'x')
# Out: True
```

Esto se hizo originalmente para poder ordenar una lista de tipos mixtos y los objetos se agruparían por tipo:

```
l = [7, 'x', (1, 2), [5, 6], 5, 8.0, 'y', 1.2, [7, 8], 'z']
sorted(l)
# Out: [1.2, 5, 7, 8.0, [5, 6], [7, 8], 'x', 'y', 'z', (1, 2)]
```

Python 3.x 3.0

Se produce una excepción al comparar diferentes tipos (no numéricos):

```
1 < 1.5
# Out: True

[1, 2] > 'foo'
# TypeError: unorderable types: list() > str()
(1, 2) > 'foo'
# TypeError: unorderable types: tuple() > str()
[1, 2] > (1, 2)
# TypeError: unorderable types: list() > tuple()
```

Para ordenar las listas mixtas en Python 3 por tipos y para lograr la compatibilidad entre versiones, debe proporcionar una clave para la función ordenada:

```
>>> list = [1, 'hello', [3, 4], {'python': 2}, 'stackoverflow', 8, {'python': 3}, [5, 6]]
>>> sorted(list, key=str)
# Out: [1, 8, [3, 4], [5, 6], 'hello', 'stackoverflow', {'python': 2}, {'python': 3}]
```

El uso de `str` como la función de `key` convierte temporalmente cada elemento en una cadena solo para fines de comparación. Luego ve la representación de la cadena comenzando con `[` , `'` , `{` o `0-9` y es capaz de ordenarlas (y todos los siguientes caracteres).

Entrada del usuario

En Python 2, la entrada del usuario se acepta utilizando la función `raw_input` ,

Python 2.x 2.3

```
user_input = raw_input()
```

Mientras que en Python 3, la entrada del usuario se acepta usando la función de `input` .

Python 3.x 3.0

```
user_input = input()
```

En Python 2, la `input` la función aceptará la entrada e *interpretarla.* Si bien esto puede ser útil, tiene varias consideraciones de seguridad y se eliminó en Python 3. Para acceder a la misma funcionalidad, se puede usar `eval(input())` .

Para mantener un script portátil en las dos versiones, puede colocar el código a continuación cerca de la parte superior de su script de Python:

```
try:
    input = raw_input
except NameError:
    pass
```

Cambios en el método del diccionario

En Python 3, muchos de los métodos del diccionario tienen un comportamiento bastante diferente al de Python 2, y muchos también fueron eliminados: `has_key` , `iter*` y `view*` se han ido. En lugar de `d.has_key(key)` , que había estado en desuso durante mucho tiempo, ahora se debe usar la `key in d` .

En Python 2, las `keys` métodos de diccionario, `values` y `items` devuelven listas. En Python 3 devuelven los objetos de *vista en su* lugar; los objetos de vista no son iteradores, y se diferencian de ellos de dos maneras, a saber:

- tienen tamaño (uno puede usar la función `len` en ellos)
- Se pueden iterar varias veces.

Además, como con los iteradores, los cambios en el diccionario se reflejan en los objetos de vista.

Python 2.7 ha respaldado estos métodos desde Python 3; están disponibles como `viewkeys` , `viewvalues` y elementos de `viewitems` . Para transformar el código de Python 2 en el código de Python 3, los formularios correspondientes son:

- `d.keys()` , `d.values()` y `d.items()` de Python 2 deben cambiarse a `list(d.keys())` , `list(d.values())` y `list(d.items())`
- `d.iterkeys()` , `d.itervalues()` y `d.iteritems()` deben cambiarse a `iter(d.keys())` , o incluso mejor, `iter(d)` ; `iter(d.values())` e `iter(d.items())` respectivamente
- y, finalmente, las llamadas al método Python 2.7 `d.viewkeys()` , `d.viewvalues()` y `d.viewitems()` se pueden reemplazar por `d.keys()` , `d.values()` y `d.items()` .

Portar el código de Python 2 que *itera* sobre las claves del diccionario, los valores o los elementos mientras se muta a veces es complicado. Considerar:

```
d = {'a': 0, 'b': 1, 'c': 2, '!': 3}
for key in d.keys():
    if key.isalpha():
        del d[key]
```

El código parece que funcionaría de manera similar en Python 3, pero allí el método de las `keys` devuelve un objeto de vista, no una lista, y si el diccionario cambia de tamaño mientras se repite, el código de Python 3 se bloqueará con `RuntimeError: dictionary changed size during iteration` . La solución es, por supuesto, escribir correctamente `for key in list(d)` .

De manera similar, los objetos de vista se comportan de manera diferente a los iteradores: uno no puede usar `next()` en ellos, y uno no puede *reanudar la* iteración; en su lugar se reiniciaría; Si el código de Python 2 pasa el valor de retorno de `d.iterkeys()` , `d.itervalues()` o `d.iteritems()` a un método que espera un iterador en lugar de un *iterable* , entonces debería ser `iter(d)` , `iter(d.values())` o `iter(d.items())` en Python 3.

La sentencia exec es una función en Python 3

En Python 2, `exec` es una declaración, con una sintaxis especial: `exec code [in globals[, locals]]` . En Python 3, `exec` ahora es una función: `exec(code, [, globals[, locals]])` , y la sintaxis de Python 2 generará un `SyntaxError` .

A medida que se cambió la `print` de una declaración a una función, también se agregó una importación `__future__` . Sin embargo, no hay `from __future__ import exec_function` , ya que no es necesario: la declaración exec en Python 2 también se puede usar con una sintaxis que se ve exactamente como la invocación de la función `exec` en Python 3. Por lo tanto, puede cambiar las declaraciones

Python 2.x 2.3

```
exec 'code'
exec 'code' in global_vars
exec 'code' in global_vars, local_vars
```

a las formas

Python 3.x 3.0

```
exec('code')
exec('code', global_vars)
exec('code', global_vars, local_vars)
```

y se garantiza que las últimas formas funcionarán de manera idéntica tanto en Python 2 como en Python 3.

Error de la función hasattr en Python 2

En Python 2, cuando una propiedad `hasattr` un error, `hasattr` ignorará esta propiedad y devolverá `False` .

```
class A(object):
    @property
    def get(self):
        raise IOError
```

```
class B(object):
    @property
    def get(self):
        return 'get in b'

a = A()
b = B()

print 'a hasattr get: ', hasattr(a, 'get')
# output False in Python 2 (fixed, True in Python 3)
print 'b hasattr get', hasattr(b, 'get')
# output True in Python 2 and Python 3
```

Este error se corrige en Python3. Así que si usas Python 2, usa

```
try:
    a.get
except AttributeError:
    print("no get property!")
```

o use `getattr` en `getattr` lugar

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("no get property!")
```

Módulos renombrados

Algunos módulos en la biblioteca estándar han sido renombrados:

Viejo nombre	Nuevo nombre
_winreg	Winreg
ConfigParser	configparser
copy_reg	copyreg
Cola	cola
SocketServer	socketserver
_markupbase	markupbase
reprimir	reprender
test.test_support	test.support
Tkinter	tkinter

Viejo nombre	Nuevo nombre
tkFileDialog	tkinter.filedialog
urllib / urllib2	urllib, urllib.parse, urllib.error, urllib.response, urllib.request, urllib.robotparser

Algunos módulos incluso se han convertido de archivos a bibliotecas. Tome tkinter y urllib desde arriba como ejemplo.

Compatibilidad

Al mantener la compatibilidad entre las versiones de Python 2.xy 3.x, puede usar el paquete externo `future` para habilitar la importación de paquetes de biblioteca de nivel superior con nombres de Python 3.x en las versiones de Python 2.x.

Constantes octales

En Python 2, un literal octal podría definirse como

```
>>> 0755  # only Python 2
```

Para asegurar la compatibilidad cruzada, use

```
0o755  # both Python 2 and Python 3
```

Todas las clases son "clases de nuevo estilo" en Python 3.

En Python `3.x` todas las clases son clases de *nuevo estilo* ; Al definir una nueva clase, Python implícitamente la hace heredar de un `object` . Como tal, especificar un `object` en una definición de `class` es completamente opcional:

Python 3.x 3.0

```
class X: pass
class Y(object): pass
```

Ambas de estas clases ahora contienen `object` en su `mro` (orden de resolución de métodos):

Python 3.x 3.0

```
>>> X.__mro__
(__main__.X, object)

>>> Y.__mro__
(__main__.Y, object)
```

En Python `2.x` clases son, por defecto, las clases de estilo antiguo; no heredan implícitamente del

`object` . Esto hace que la semántica de las clases difiera dependiendo de si agregamos explícitamente el `object` como una `class` base:

Python 2.x 2.3

```
class X: pass
class Y(object): pass
```

En este caso, si intentamos imprimir el `__mro__` de `Y` , `__mro__` una salida similar a la del caso Python `3.x` :

Python 2.x 2.3

```
>>> Y.__mro__
(<class '__main__.Y'>, <type 'object'>)
```

Esto sucede porque hicimos que `Y` heredara explícitamente de un objeto al definirlo: `class Y(object): pass` . Para la clase `X` que *no* hereda del objeto, el atributo `__mro__` no existe, al intentar acceder a él se obtiene un `AttributeError` .

Para **garantizar la compatibilidad** entre ambas versiones de Python, las clases se pueden definir con el `object` como una clase base:

```
class mycls(object):
    """I am fully compatible with Python 2/3"""
```

Alternativamente, si la variable `__metaclass__` está configurada para `type` en el ámbito global, todas las clases definidas posteriormente en un módulo dado son implícitamente un estilo nuevo sin la necesidad de heredar explícitamente del `object` :

```
__metaclass__ = type

class mycls:
    """I am also fully compatible with Python 2/3"""
```

Se eliminaron los operadores <> y ``, sinónimo de! = Y repr ()

En Python 2, `<>` es un sinónimo para `!=` ; Del mismo modo, `` `foo` `` es un sinónimo de `repr(foo)` .

Python 2.x 2.7

```
>>> 1 <> 2
True
>>> 1 <> 1
False
>>> foo = 'hello world'
>>> repr(foo)
"'hello world'"
>>> `foo`
"'hello world'"
```

Python 3.x 3.0

```
>>> 1 <> 2
  File "<stdin>", line 1
    1 <> 2
       ^
SyntaxError: invalid syntax
>>> `foo`
  File "<stdin>", line 1
    `foo`
    ^
SyntaxError: invalid syntax
```

codificar / decodificar a hex ya no está disponible

Python 2.x 2.7

```
"1deadbeef3".decode('hex')
# Out: '\x1d\xea\xdb\xee\xf3'
'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Out: 1deadbeef3
```

Python 3.x 3.0

```
"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: 'str' object has no attribute 'decode'

b"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.decode() to handle arbitrary codecs

'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.encode() to handle arbitrary codecs

b'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# AttributeError: 'bytes' object has no attribute 'encode'
```

Sin embargo, como lo sugiere el mensaje de error, puede usar el módulo de `codecs` para lograr el mismo resultado:

```
import codecs
codecs.decode('1deadbeef4', 'hex')
# Out: b'\x1d\xea\xdb\xee\xf4'
codecs.encode(b'\x1d\xea\xdb\xee\xf4', 'hex')
# Out: b'1deadbeef4'
```

Tenga en cuenta que `codecs.encode` devuelve un objeto de `bytes` . Para obtener un objeto `str` simplemente `decode` a ASCII:

```
codecs.encode(b'\x1d\xea\xdb\xee\xff', 'hex').decode('ascii')
```

```
# Out: '1deadbeeff'
```

Función cmp eliminada en Python 3

En Python 3 se eliminó la función incorporada de `cmp` , junto con el método especial `__cmp__` .

De la documentación:

> La función `cmp()` debe tratar como pasada, y el método especial `__cmp__()` ya no se admite. Use `__lt__()` para clasificar, `__eq__()` con `__hash__()` , y otras comparaciones ricas según sea necesario. (Si realmente necesita la funcionalidad `cmp()` , puede usar la expresión `(a > b) - (a < b)` como equivalente para `cmp(a, b)` .

Además, todas las funciones incorporadas que aceptaron el parámetro `cmp` ahora solo aceptan el parámetro de `key` clave única.

En el módulo `functools` también hay una función útil `cmp_to_key(func)` que le permite convertir de una función de estilo `cmp` a una `key` estilo de `key` :

> Transformar una función de comparación de estilo antiguo en una función clave. Se usa con herramientas que aceptan funciones clave (como `sorted()` , `min()` , `max()` , `heapq.nlargest()` , `heapq.nsmallest()` , `itertools.groupby()`). Esta función se utiliza principalmente como una herramienta de transición para los programas que se convierten desde Python 2 que admite el uso de funciones de comparación.

Variables filtradas en la lista de comprensión.

Python 2.x 2.3

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'U'
```

Python 3.x 3.0

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'hello world!'
```

Como se puede ver en el ejemplo, en Python 2 se filtró el valor de `x` : ¡enmascaró a `hello world!` e imprimió `U` , ya que este fue el último valor de `x` cuando finalizó el bucle.

Sin embargo, en Python 3 `x` imprime el `hello world!` originalmente definido `hello world!` , ya que la

variable local de la lista de comprensión no enmascara las variables del ámbito circundante.

Además, ni las expresiones generadoras (disponibles en Python desde la versión 2.5) ni las comprensiones de diccionario o conjunto (que fueron respaldadas a Python 2.7 desde Python 3) filtran las variables en Python 2.

Tenga en cuenta que tanto en Python 2 como en Python 3, las variables se filtrarán en el ámbito circundante cuando se use un bucle for:

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
print(x)
# Out: 'U'
```

mapa()

`map()` es un componente que es útil para aplicar una función a elementos de un iterable. En Python 2, el `map` devuelve una lista. En Python 3, `map` devuelve un *objeto map* , que es un generador.

```
# Python 2.X
>>> map(str, [1, 2, 3, 4, 5])
['1', '2', '3', '4', '5']
>>> type(_)
>>> <class 'list'>

# Python 3.X
>>> map(str, [1, 2, 3, 4, 5])
<map object at 0x*>
>>> type(_)
<class 'map'>

# We need to apply map again because we "consumed" the previous map....
>>> map(str, [1, 2, 3, 4, 5])
>>> list(_)
['1', '2', '3', '4', '5']
```

En Python 2, puede pasar `None` para que funcione como una función de identidad. Esto ya no funciona en Python 3.

Python 2.x 2.3

```
>>> map(None, [0, 1, 2, 3, 0, 4])
[0, 1, 2, 3, 0, 4]
```

Python 3.x 3.0

```
>>> list(map(None, [0, 1, 2, 3, 0, 5]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

Además, al pasar más de un iterable como argumento en Python 2, el `map` rellena los iterables más cortos con `None` (similar a `itertools.izip_longest`). En Python 3, la iteración se detiene después de la iteración más corta.

En Python 2:

Python 2.x 2.3

```
>>> map(None, [1, 2, 3], [1, 2], [1, 2, 3, 4, 5])
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

En Python 3:

Python 3.x 3.0

```
>>> list(map(lambda x, y, z: (x, y, z), [1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2)]

# to obtain the same padding as in Python 2 use zip_longest from itertools
>>> import itertools
>>> list(itertools.zip_longest([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

Nota : en lugar de un `map` considere utilizar listas de comprensión, que son compatibles con Python 2/3. Reemplazo del `map(str, [1, 2, 3, 4, 5])` :

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

filter (), map () y zip () devuelven iteradores en lugar de secuencias

Python 2.x 2.7

En el `filter` Python 2, las funciones incorporadas de `map` y `zip` devuelven una secuencia. `map` y `zip` siempre devuelven una lista, mientras que con el `filter` el tipo de retorno depende del tipo de parámetro dado:

```
>>> s = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> s
'abc'
>>> s = map(lambda x: x * x, [0, 1, 2])
>>> s
[0, 1, 4]
>>> s = zip([0, 1, 2], [3, 4, 5])
>>> s
[(0, 3), (1, 4), (2, 5)]
```

Python 3.x 3.0

En el `filter` Python 3, el `map` y el `zip` iterador de retorno en su lugar:

```
>>> it = filter(lambda x: x.isalpha(), 'a1b2c3')
```

```
>>> it
<filter object at 0x00000098A55C2518>
>>> ''.join(it)
'abc'
>>> it = map(lambda x: x * x, [0, 1, 2])
>>> it
<map object at 0x000000E0763C2D30>
>>> list(it)
[0, 1, 4]
>>> it = zip([0, 1, 2], [3, 4, 5])
>>> it
<zip object at 0x000000E0763C52C8>
>>> list(it)
[(0, 3), (1, 4), (2, 5)]
```

Desde Python 2, `itertools.izip` es equivalente a Python 3 `zip` `izip` se ha eliminado en Python 3.

Importaciones absolutas / relativas

En Python 3, PEP 404 cambia la forma en que funcionan las importaciones desde Python 2. Ya no se permiten las importaciones *relativas implícitas* en los paquetes y `from ... import *` solo se permiten en el código de nivel de módulo.

Para lograr el comportamiento de Python 3 en Python 2:

- la característica de importaciones absolutas se puede habilitar `from __future__ import absolute_import`
- Se alientan las importaciones *relativas explícitas* en lugar de *las* importaciones *relativas implícitas*

Para aclarar, en Python 2, un módulo puede importar el contenido de otro módulo ubicado en el mismo directorio de la siguiente manera:

```
import foo
```

Observe que la ubicación de `foo` es ambigua desde la declaración de importación solo. Este tipo de importación relativa implícita se desaconseja, por tanto, a favor de las importaciones relativas explícitas , que se parecen a las siguientes:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
from ...package import bar
from ...sys import path
```

El punto `.` Permite una declaración explícita de la ubicación del módulo dentro del árbol de directorios.

Más sobre Importaciones Relativas

Considere algún paquete definido por el usuario llamado `shapes` . La estructura del directorio es la siguiente:

```
shapes
├── __init__.py
|
├── circle.py
|
├── square.py
|
└── triangle.py
```

`circle.py` , `square.py` y `triangle.py` importan `util.py` como un módulo. ¿Cómo se referirán a un módulo en el mismo nivel?

```
 from . import util # use util.PI, util.sq(x), etc
```

O

```
 from .util import * #use PI, sq(x), etc to call functions
```

El . Se utiliza para importaciones relativas del mismo nivel.

Ahora, considere un diseño alternativo del módulo de `shapes` :

```
shapes
├── __init__.py
|
├── circle
|   ├── __init__.py
|   └── circle.py
|
├── square
|   ├── __init__.py
|   └── square.py
|
├── triangle
|   ├── __init__.py
|   ├── triangle.py
|
└── util.py
```

Ahora, ¿cómo se referirán estas 3 clases a util.py?

```
 from .. import util # use util.PI, util.sq(x), etc
```

O

```
 from ..util import * # use PI, sq(x), etc to call functions
```

El `..` se utiliza para las importaciones relativas de nivel padre. Añadir más `.` s con número de niveles entre el padre y el niño.

Archivo I / O

`file` ya no es un nombre incorporado en 3.x (`open` aún funciona).

Los detalles internos del archivo de E / S se han trasladado al módulo de la biblioteca estándar `io` , que también es el nuevo hogar de `StringIO` :

```
import io
assert io.open is open # the builtin is an alias
buffer = io.StringIO()
buffer.write('hello, ') # returns number of characters written
buffer.write('world!\n')
buffer.getvalue() # 'hello, world!\n'
```

El modo de archivo (texto frente a binario) ahora determina el tipo de datos producidos al leer un archivo (y el tipo requerido para escribir):

```
with open('data.txt') as f:
    first_line = next(f)
    assert type(first_line) is str
with open('data.bin', 'rb') as f:
    first_kb = f.read(1024)
    assert type(first_kb) is bytes
```

La codificación de los archivos de texto se establece de forma predeterminada en `locale.getpreferredencoding(False)` . Para especificar una codificación explícitamente, use el parámetro de palabra clave de `encoding` :

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

La función round () rompe el empate y devuelve el tipo

rotura de corbata redonda ()

En Python 2, usar `round()` en un número igualmente cercano a dos enteros devolverá el más alejado de 0. Por ejemplo:

Python 2.x 2.7

```
round(1.5)  # Out: 2.0
round(0.5)  # Out: 1.0
round(-0.5)  # Out: -1.0
round(-1.5)  # Out: -2.0
```

Sin embargo, en Python 3, `round()` devolverá el entero par (también conocido *como redondeo de banqueros*). Por ejemplo:

Python 3.x 3.0

```
round(1.5)  # Out: 2
round(0.5)  # Out: 0
round(-0.5)  # Out: 0
round(-1.5)  # Out: -2
```

La función round () sigue la *mitad de la* estrategia de *redondeo uniforme que redondeará los* números a mitad de camino al entero par más cercano (por ejemplo, `round(2.5)` ahora devuelve 2 en lugar de 3.0).

Según la referencia en Wikipedia , esto también se conoce como *redondeo imparcial* , *redondeo convergente* , *redondeo estadístico, redondeo holandés* , *redondeo gaussiano* o *redondeo impar* .

La mitad del redondeo uniforme es parte del estándar IEEE 754 y también es el modo de redondeo predeterminado en .NET de Microsoft.

Esta estrategia de redondeo tiende a reducir el error de redondeo total. Como en promedio, la cantidad de números que se redondean es la misma que la cantidad de números que se redondean hacia abajo, los errores de redondeo se cancelan. En cambio, otros métodos de redondeo tienden a tener un sesgo hacia arriba o hacia abajo en el error promedio.

---

round () tipo de retorno

La función `round()` devuelve un tipo `float` en Python 2.7

Python 2.x 2.7

```
round(4.8)
# 5.0
```

A partir de Python 3.0, si se omite el segundo argumento (número de dígitos), devuelve un `int` .

Python 3.x 3.0

```
round(4.8)
# 5
```

Verdadero, Falso y Ninguno

En Python 2, `True` , `False` y `None` son constantes integradas. Lo que significa que es posible reasignarlos.

Python 2.x 2.0

```
True, False = False, True
True   # False
False  # True
```

No puedes hacer esto con `None` desde Python 2.4.

Python 2.x 2.4

```
None = None  # SyntaxError: cannot assign to None
```

En Python 3, `True` , `False` y `None` ahora son palabras clave.

Python 3.x 3.0

```
True, False = False, True  # SyntaxError: can't assign to keyword

None = None  # SyntaxError: can't assign to keyword
```

Devolver valor al escribir en un objeto de archivo

En Python 2, escribir directamente en un identificador de archivo devuelve `None` :

Python 2.x 2.3

```
hi = sys.stdout.write('hello world\n')
# Out: hello world
type(hi)
# Out: <type 'NoneType'>
```

En Python 3, escribir en un identificador devolverá el número de caracteres escritos al escribir texto, y el número de bytes escritos al escribir bytes:

Python 3.x 3.0

```
import sys

char_count = sys.stdout.write('hello world \n')
# Out: hello world 
char_count
# Out: 14

byte_count = sys.stdout.buffer.write(b'hello world \xf0\x9f\x90\x8d\n')
# Out: hello world 
byte_count
# Out: 17
```

largo vs. int

En Python 2, cualquier entero mayor que un C `ssize_t` se convertiría en el tipo de datos `long` , indicado por un sufijo `L` en el literal. Por ejemplo, en una compilación de Python de 32 bits:

Python 2.x 2.7

```
>>> 2**31
2147483648L
>>> type(2**31)
<type 'long'>
>>> 2**30
1073741824
>>> type(2**30)
```

```
<type 'int'>
>>> 2**31 - 1  # 2**31 is long and long - int is long
2147483647L
```

Sin embargo, en Python 3, el tipo de datos `long` fue eliminado; no importa qué tan grande sea el número entero, será un `int` .

Python 3.x 3.0

```
2**1024
# Output:
17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602112

print(-(2**1024))
# Output: -
17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602112

type(2**1024)
# Output: <class 'int'>
```

Valor booleano de clase

Python 2.x 2.7

En Python 2, si desea definir un valor booleano de clase por sí mismo, debe implementar el método `__nonzero__` en su clase. El valor es True por defecto.

```
class MyClass:
    def __nonzero__(self):
        return False

my_instance = MyClass()
print bool(MyClass)       # True
print bool(my_instance)   # False
```

Python 3.x 3.0

En Python 3, `__bool__` se usa en lugar de `__nonzero__`

```
class MyClass:
    def __bool__(self):
        return False

my_instance = MyClass()
print(bool(MyClass))       # True
print(bool(my_instance))   # False
```

Lea Incompatibilidades que se mueven de Python 2 a Python 3 en línea: https://riptutorial.com/es/python/topic/809/incompatibilidades-que-se-mueven-de-python-2-a-python-3

Capítulo 101: Indexación y corte

Sintaxis

- obj [inicio: detener: paso]
- cortar
- rebanada (inicio, parada [, paso])

Parámetros

Paramer	Descripción
`obj`	El objeto del que desea extraer un "subobjeto" de
`start`	El índice de `obj` que desea que comience el subobjeto (tenga en cuenta que Python tiene un índice de cero, lo que significa que el primer elemento de `obj` tiene un índice de `0`). Si se omite, el valor predeterminado es `0` .
`stop`	El índice (no incluido) de `obj` que desea que termine el subobjeto. Si se omite, el valor predeterminado es `len(obj)` .
`step`	Le permite seleccionar solo cada elemento de `step` . Si se omite, el valor predeterminado es `1` .

Observaciones

Puede unificar el concepto de corte de cadenas con el de cortar otras secuencias al ver las cadenas como una colección de caracteres inmutables, con la advertencia de que un carácter Unicode está representado por una cadena de longitud 1.

En la notación matemática, puede considerar dividir para usar un intervalo semiabierto de `[start, end)` , es decir, que el inicio está incluido pero el final no. La naturaleza semiabierta del intervalo tiene la ventaja de que `len(x[:n])` = `n` donde `len(x)` > = `n` , mientras que el intervalo que se cierra al inicio tiene la ventaja de que `x[n:n+1]` = `[x[n]]` donde `x` es una lista con `len(x) >= n` , manteniendo así la coherencia entre la notación de indexación y de corte.

Examples

Rebanado basico

Para cualquier iterable (por ejemplo, una cadena, lista, etc.), Python le permite dividir y devolver una subcadena o una lista secundaria de sus datos.

Formato para rebanar:

```
iterable_name[start:stop:step]
```

dónde,

- `start` es el primer índice de la rebanada. Por defecto es 0 (el índice del primer elemento)
- `stop` uno más allá del último índice de la rebanada. Por defecto a len (iterable)
- `step` es el tamaño del paso (mejor explicado por los ejemplos a continuación)

Ejemplos:

```
a = "abcdef"
a             # "abcdef"
              # Same as a[:] or a[::] since it uses the defaults for all three indices
a[-1]         # "f"
a[:]          # "abcdef"
a[::]         # "abcdef"
a[3:]         # "def" (from index 3, to end(defaults to size of iterable))
a[:4]         # "abcd" (from beginning(default 0) to position 4 (excluded))
a[2:4]        # "cd" (from position 2, to position 4 (excluded))
```

Además, cualquiera de los anteriores se puede utilizar con el tamaño de paso definido:

```
a[::2]        # "ace" (every 2nd element)
a[1:4:2]      # "bd" (from index 1, to index 4 (excluded), every 2nd element)
```

Los índices pueden ser negativos, en cuyo caso se calculan desde el final de la secuencia.

```
a[:-1]      # "abcde" (from index 0 (default), to the second last element (last element - 1))
a[:-2]      # "abcd" (from index 0 (default), to the third last element (last element -2))
a[-1:]      # "f" (from the last element to the end (default len())
```

Los tamaños de los pasos también pueden ser negativos, en cuyo caso la división se repetirá en orden inverso:

```
a[3:1:-1]   # "dc" (from index 2 to None (default), in reverse order)
```

Este constructo es útil para revertir un iterable.

```
a[::-1]     # "fedcba" (from last element (default len()-1), to first, in reverse order(-1))
```

Tenga en cuenta que para los pasos negativos, el `end_index` predeterminado es `None` (consulte http://stackoverflow.com/a/12521981)

```
a[5:None:-1] # "fedcba" (this is equivalent to a[::-1])
a[5:0:-1]    # "fedcb" (from the last element (index 5) to second element (index 1)
```

Hacer una copia superficial de una matriz

Una forma rápida de hacer una copia de una matriz (en lugar de asignar una variable con otra

referencia a la matriz original) es:

```
arr[:]
```

Vamos a examinar la sintaxis. `[:]` Medios que `start` , `end` , y `slice` están omitidos. Los valores predeterminados son `0` , `len(arr)` y `1` , respectivamente, lo que significa que el subarreglo que solicitamos tendrá todos los elementos de `arr` desde el principio hasta el final.

En la práctica, esto se parece a algo como:

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)    # ['a', 'b', 'c', 'd']
print(copy)   # ['a', 'b', 'c']
```

Como puede ver, `arr.append('d')` agregó `d` a `arr` , ¡pero la `copy` mantuvo sin cambios!

Tenga en cuenta que esto hace una copia *superficial* y es idéntico a `arr.copy()` .

Invertir un objeto

Puede usar cortes para revertir muy fácilmente una `str` , `list` o `tuple` (o básicamente cualquier objeto de colección que implemente el corte con el parámetro de paso). Este es un ejemplo de inversión de una cadena, aunque esto se aplica igualmente a los otros tipos mencionados anteriormente:

```
s = 'reverse me!'
s[::-1]    # '!em esrever'
```

Veamos rápidamente la sintaxis. `[::-1]` significa que la división debe ser desde el principio hasta el final de la cadena (porque se omiten el `start` y el `end`) y un paso de `-1` significa que debe moverse a través de la cadena en sentido inverso.

Clases personalizadas de indexación: __getitem__, __setitem__ y __delitem__

```
class MultiIndexingList:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return repr(self.value)

    def __getitem__(self, item):
        if isinstance(item, (int, slice)):
            return self.__class__(self.value[item])
        return [self.value[i] for i in item]

    def __setitem__(self, item, value):
        if isinstance(item, int):
            self.value[item] = value
        elif isinstance(item, slice):
```

```
            raise ValueError('Cannot interpret slice with multiindexing')
        else:
            for i in item:
                if isinstance(i, slice):
                    raise ValueError('Cannot interpret slice with multiindexing')
                self.value[i] = value

    def __delitem__(self, item):
        if isinstance(item, int):
            del self.value[item]
        elif isinstance(item, slice):
            del self.value[item]
        else:
            if any(isinstance(elem, slice) for elem in item):
                raise ValueError('Cannot interpret slice with multiindexing')
            item = sorted(item, reverse=True)
            for elem in item:
                del self.value[elem]
```

Esto permite el corte y la indexación para el acceso al elemento:

```
a = MultiIndexingList([1,2,3,4,5,6,7,8])
a
# Out: [1, 2, 3, 4, 5, 6, 7, 8]
a[1,5,2,6,1]
# Out: [2, 6, 3, 7, 2]
a[4, 1, 5:, 2, ::2]
# Out: [5, 2, [6, 7, 8], 3, [1, 3, 5, 7]]
#       4|1-|----50:---|2-|-----::2-----   <-- indicated which element came from which index
```

Al configurar y eliminar elementos solo se permite la indexación de enteros *separados* por *comas* (sin rebanar):

```
a[4] = 1000
a
# Out: [1, 2, 3, 4, 1000, 6, 7, 8]
a[2,6,1] = 100
a
# Out: [1, 100, 100, 4, 1000, 6, 100, 8]
del a[5]
a
# Out: [1, 100, 100, 4, 1000, 100, 8]
del a[4,2,5]
a
# Out: [1, 100, 4, 8]
```

Asignación de rebanada

Otra característica interesante que utiliza los segmentos es la asignación de sectores. Python le permite asignar nuevos segmentos para reemplazar los segmentos antiguos de una lista en una sola operación.

Esto significa que si tiene una lista, puede reemplazar varios miembros en una sola tarea:

```
lst = [1, 2, 3]
```

```
lst[1:3] = [4, 5]
print(lst) # Out: [1, 4, 5]
```

La asignación tampoco debe coincidir en tamaño, por lo que si desea reemplazar una porción antigua por una nueva que es de tamaño diferente, podría:

```
lst = [1, 2, 3, 4, 5]
lst[1:4] = [6]
print(lst) # Out: [1, 6, 5]
```

También es posible usar la sintaxis de corte conocida para hacer cosas como reemplazar toda la lista:

```
lst = [1, 2, 3]
lst[:] = [4, 5, 6]
print(lst) # Out: [4, 5, 6]
```

O solo los dos últimos miembros:

```
lst = [1, 2, 3]
lst[-2:] = [4, 5, 6]
print(lst) # Out: [1, 4, 5, 6]
```

Rebanar objetos

Los cortes son objetos en sí mismos y se pueden almacenar en variables con la función de `slice()` incorporada. Las variables de división se pueden usar para hacer que su código sea más legible y para promover la reutilización.

```
>>> programmer_1 = [ 1956, 'Guido', 'van Rossum', 'Python', 'Netherlands']
>>> programmer_2 = [ 1815, 'Ada', 'Lovelace', 'Analytical Engine', 'England']
>>> name_columns = slice(1, 3)
>>> programmer_1[name_columns]
['Guido', 'van Rossum']
>>> programmer_2[name_columns]
['Ada', 'Lovelace']
```

Indexación básica

Las listas de Python están basadas en 0, *es decir,* se puede acceder al primer elemento de la lista mediante el índice `0`

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

Puede acceder al segundo elemento de la lista por índice `1` , tercer elemento por índice `2` y así sucesivamente:

```
print(arr[1])
```

```
>> 'b'
print(arr[2])
>> 'c'
```

También puede usar índices negativos para acceder a los elementos del final de la lista. p.ej. El índice `-1` le dará el último elemento de la lista y el índice `-2` le dará el segundo elemento de la lista:

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

Si intenta acceder a un índice que no está presente en la lista, se `IndexError` un `IndexError` :

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Lea Indexación y corte en línea: https://riptutorial.com/es/python/topic/289/indexacion-y-corte

Capítulo 102: Interfaz de puerta de enlace de servidor web (WSGI)

Parámetros

Parámetro	Detalles
start_response	Una función utilizada para procesar el inicio.

Examples

Objeto de servidor (Método)

A nuestro objeto de servidor se le asigna un parámetro de 'aplicación' que puede ser cualquier objeto de aplicación que se pueda llamar (ver otros ejemplos). Primero escribe los encabezados, luego el cuerpo de datos devueltos por nuestra aplicación a la salida estándar del sistema.

```
import os, sys

def run(application):
    environ['wsgi.input']        = sys.stdin
    environ['wsgi.errors']       = sys.stderr

    headers_set = []
    headers_sent = []

    def write (data):
        """
        Writes header data from 'start_response()' as well as body data from 'response'
        to system standard output.
        """
        if not headers_set:
            raise AssertionError("write() before start_response()")

        elif not headers_sent:
            status, response_headers = headers_sent[:] = headers_set
            sys.stdout.write('Status: %s\r\n' % status)
            for header in response_headers:
                sys.stdout.write('%s: %s\r\n' % header)
            sys.stdout.write('\r\n')

        sys.stdout.write(data)
        sys.stdout.flush()

    def start_response(status, response_headers):
        """ Sets headers for the response returned by this server."""
        if headers_set:
            raise AssertionError("Headers already set!")

        headers_set[:] = [status, response_headers]
        return write
```

```
    # This is the most important piece of the 'server object'
    # Our result will be generated by the 'application' given to this method as a parameter
    result = application(environ, start_response)
    try:
        for data in result:
            if data:
                write(data)          # Body isn't empty send its data to 'write()'
        if not headers_sent:
            write('')                # Body is empty, send empty string to 'write()'
```

Lea Interfaz de puerta de enlace de servidor web (WSGI) en línea:
https://riptutorial.com/es/python/topic/5315/interfaz-de-puerta-de-enlace-de-servidor-web--wsgi-

Capítulo 103: Introducción a RabbitMQ utilizando AMQPStorm

Observaciones

La última versión de AMQPStorm está disponible en pypi o puede instalarla usando pip

```
pip install amqpstorm
```

Examples

Cómo consumir mensajes de RabbitMQ

Comience con la importación de la biblioteca.

```
from amqpstorm import Connection
```

Al consumir mensajes, primero debemos definir una función para manejar los mensajes entrantes. Esta puede ser cualquier función que se pueda llamar y tiene que tomar un objeto de mensaje o una tupla de mensaje (según el parámetro `to_tuple` definido en `start_consuming`).

Además de procesar los datos del mensaje entrante, también tendremos que Reconocer o Rechazar el mensaje. Esto es importante, ya que necesitamos informar a RabbitMQ que recibimos y procesamos el mensaje correctamente.

```
def on_message(message):
    """This function is called on message received.

    :param message: Delivered message.
    :return:
    """
    print("Message:", message.body)

    # Acknowledge that we handled the message without any issues.
    message.ack()

    # Reject the message.
    # message.reject()

    # Reject the message, and put it back in the queue.
    # message.reject(requeue=True)
```

A continuación, debemos configurar la conexión al servidor RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

Después de eso tenemos que configurar un canal. Cada conexión puede tener múltiples canales

y, en general, al realizar tareas de subprocesos múltiples, se recomienda (pero no es obligatorio) tener uno por subproceso.

```
channel = connection.channel()
```

Una vez que tengamos configurado nuestro canal, debemos informarle a RabbitMQ que queremos comenzar a consumir mensajes. En este caso, usaremos nuestra función `on_message` previamente definida para manejar todos nuestros mensajes consumidos.

La cola que escucharemos en el servidor RabbitMQ será `simple_queue` , y también le estamos diciendo a RabbitMQ que reconoceremos todos los mensajes entrantes una vez que hayamos terminado con ellos.

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

Finalmente, necesitamos iniciar el bucle IO para comenzar a procesar los mensajes entregados por el servidor RabbitMQ.

```
channel.start_consuming(to_tuple=False)
```

Cómo publicar mensajes a RabbitMQ

Comience con la importación de la biblioteca.

```
from amqpstorm import Connection
from amqpstorm import Message
```

Luego necesitamos abrir una conexión al servidor RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

Después de eso tenemos que configurar un canal. Cada conexión puede tener múltiples canales y, en general, al realizar tareas de subprocesos múltiples, se recomienda (pero no es obligatorio) tener uno por subproceso.

```
channel = connection.channel()
```

Una vez que tengamos configurado nuestro canal, podemos comenzar a preparar nuestro mensaje.

```
# Message Properties.
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
}

# Create the message.
message = Message.create(channel=channel, body='Hello World!', properties=properties)
```

Ahora podemos publicar el mensaje simplemente llamando a `publish` y proporcionando una `routing_key` . En este caso, vamos a enviar el mensaje a una cola llamada `simple_queue` .

```
message.publish(routing_key='simple_queue')
```

Cómo crear una cola retrasada en RabbitMQ

Primero debemos configurar dos canales básicos, uno para la cola principal y otro para la cola de demora. En mi ejemplo al final, incluyo un par de marcas adicionales que no son necesarias, pero hacen que el código sea más confiable; tales como `confirm delivery` , `delivery_mode` y `durable` . Puede encontrar más información sobre estos en el manual de RabbitMQ.

Después de configurar los canales, agregamos un enlace al canal principal que podemos utilizar para enviar mensajes desde el canal de retardo a nuestra cola principal.

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

A continuación, debemos configurar nuestro canal de retardo para reenviar los mensajes a la cola principal una vez que hayan caducado.

```
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000,
    'x-dead-letter-exchange': 'amq.direct',
    'x-dead-letter-routing-key': 'hello'
})
```

- x-message-ttl *(Mensaje - Time To Live)*

  Normalmente se usa para eliminar automáticamente los mensajes antiguos en la cola después de una duración específica, pero al agregar dos argumentos opcionales podemos cambiar este comportamiento y, en cambio, hacer que este parámetro determine en milisegundos el tiempo que los mensajes permanecerán en la cola de demora.

- x-dead-letter-routing-key

  Esta variable nos permite transferir el mensaje a una cola diferente una vez que han caducado, en lugar del comportamiento predeterminado de eliminarlo por completo.

- x-dead-letter-exchange

  Esta variable determina qué Exchange usó para transferir el mensaje de hello_delay a hello queue.

Publicación en la cola de retardo

Cuando hayamos terminado de configurar todos los parámetros básicos de Pika, simplemente envíe un mensaje a la cola de demora utilizando la publicación básica.

```
delay_channel.basic.publish(exchange='',
```

```
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mod': 2})
```

Una vez que haya ejecutado el script, debería ver las siguientes colas creadas en su módulo de administración RabbitMQ.

Overview					Messages			Messa	
Name	**Exclusive**	Parameters	**Policy**	**Status**	**Ready**	**Unacked**	**Total**	**incoming**	**deliv**
hello		D		Idle	1	0	1		
hello_delay		TTL DLX DLK D		Idle	0	0	0	0.00/s	

Ejemplo.

```
from amqpstorm import Connection

connection = Connection('127.0.0.1', 'guest', 'guest')

# Create normal 'Hello World' type channel.
channel = connection.channel()
channel.confirm_deliveries()
channel.queue.declare(queue='hello', durable=True)

# We need to bind this channel to an exchange, that will be used to transfer
# messages from our delay queue.
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')

# Create our delay channel.
delay_channel = connection.channel()
delay_channel.confirm_deliveries()

# This is where we declare the delay, and routing for our delay channel.
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000, # Delay until the message is transferred in milliseconds.
    'x-dead-letter-exchange': 'amq.direct', # Exchange used to transfer the message from A to
B.
    'x-dead-letter-routing-key': 'hello' # Name of the queue we want the message transferred
to.
})

delay_channel.basic.publish(exchange='',
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mode': 2})

print("[x] Sent")
```

Lea Introducción a RabbitMQ utilizando AMQPStorm en línea:
https://riptutorial.com/es/python/topic/3373/introduccion-a-rabbitmq-utilizando-amqpstorm

Capítulo 104: Iterables e iteradores

Examples

Iterador vs Iterable vs generador

Un **iterable** es un objeto que puede devolver un **iterador** . Cualquier objeto con estado que tenga un método `__iter__` y devuelva un iterador es iterable. También puede ser un objeto *sin* estado que implemente un método `__getitem__` . - El método puede tomar índices (comenzando desde cero) y generar un `IndexError` cuando los índices ya no son válidos.

La clase `str` de Python es un ejemplo de `__getitem__` iterable.

Un **iterador** es un objeto que produce el siguiente valor en una secuencia cuando llama al `next(*object*)` en algún objeto. Además, cualquier objeto con un método `__next__` es un iterador. Un iterador genera `StopIteration` después de agotar el iterador y *no se* puede reutilizar en este punto.

Clases iterables

Las clases `__iter__` definen un `__iter__` y `__next__` . Ejemplo de una clase iterable:

```
class MyIterable:

    def __iter__(self):

         return self

    def __next__(self):
         #code

#Classic iterable object in older versions of python, __getitem__ is still supported...
class MySequence:

    def __getitem__(self, index):

         if (condition):
             raise IndexError
         return (item)

 #Can produce a plain `iterator` instance by using iter(MySequence())
```

Intentando crear una instancia de la clase abstracta del módulo de `collections` para ver mejor esto.

Ejemplo:

Python 2.x 2.3

```
import collections
>>> collections.Iterator()
```

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods next
```

Python 3.x 3.0

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods __next__
```

Maneje la compatibilidad de Python 3 para las clases iterables en Python 2 haciendo lo siguiente:

Python 2.x 2.3

```
class MyIterable(object): #or collections.Iterator, which I'd recommend....

     ....

     def __iter__(self):

          return self

     def next(self): #code

     __next__ = next
```

Ambos son ahora iteradores y se pueden pasar a través de:

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

Los generadores son formas simples de crear iteradores. Un generador *es* un iterador y un iterador es un iterable.

Lo que puede ser iterable

Iterable puede ser cualquier cosa por la cual los artículos se reciben *uno por uno, solo hacia adelante* . Las colecciones Python incorporadas son iterables:

```
[1, 2, 3]     # list, iterate over items
(1, 2, 3)     # tuple
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict, iterate over keys
```

Los generadores devuelven iterables:

```
def foo():  # foo isn't iterable yet...
    yield 1

res = foo()  # ...but res already is
```

Iterando sobre todo iterable

```
s = {1, 2, 3}

# get every element in s
for a in s:
    print a  # prints 1, then 2, then 3

# copy into list
l1 = list(s)  # l1 = [1, 2, 3]

# use list comprehension
l2 = [a * 2 for a in s if a > 2]  # l2 = [6]
```

Verificar solo un elemento en iterable.

Utilice el desembalaje para extraer el primer elemento y asegurarse de que sea el único:

```
a, = iterable

def foo():
    yield 1

a, = foo()  # a = 1

nums = [1, 2, 3]
a, = nums  # ValueError: too many values to unpack
```

Extraer valores uno por uno.

Comience con `iter()` incorporado para obtener **iterador** sobre iterable y use `next()` para obtener elementos uno por uno hasta que se `StopIteration` para `StopIteration` el final:

```
s = {1, 2}   # or list or generator or even iterator
i = iter(s)  # get iterator
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # raises StopIteration
```

¡El iterador no es reentrante!

```
def gen():
    yield 1

iterable = gen()
for a in iterable:
    print a

# What was the first item of iterable? No way to get it now.
# Only to get a new iterator
gen()
```

Lea Iterables e iteradores en línea: https://riptutorial.com/es/python/topic/2343/iterables-e-iteradores

Capítulo 105: kivy - Framework Python multiplataforma para el desarrollo de NUI

Introducción

NUI: una interfaz de usuario natural (NUI) es un sistema para la interacción persona-computadora que el usuario opera a través de acciones intuitivas relacionadas con el comportamiento humano natural y cotidiano.

Kivy es una biblioteca de Python para el desarrollo de aplicaciones ricas en medios con capacidad multitáctil que se pueden instalar en diferentes dispositivos. La función multitáctil se refiere a la capacidad de una superficie sensible al tacto (generalmente una pantalla táctil o un trackpad) para detectar o detectar entradas provenientes de dos o más puntos de contacto simultáneamente.

Examples

Primera aplicación

Para crear una aplicación kivy.

1. subclase la clase de **aplicación**
2. Implementar el método de **construcción** , que devolverá el widget.
3. Crea una instancia de la clase e invoca la **carrera** .

```
from kivy.app import App
from kivy.uix.label import Label

class Test(App):
    def build(self):
        return Label(text='Hello world')

if __name__ == '__main__':
    Test().run()
```

Explicación

```
from kivy.app import App
```

La declaración anterior importará la **aplicación de** clase principal. Esto estará presente en su directorio de instalación directorio_instalación / kivy / app.py

```
from kivy.uix.label import Label
```

La declaración anterior importará la **etiqueta del** elemento ux. Todos los elementos ux están presentes en su directorio de instalación directorio_instalación / kivy / uix /.

```
class Test(App):
```

La declaración anterior es para crear su aplicación y el nombre de la clase será el nombre de su aplicación. Esta clase se hereda de la clase de la aplicación padre.

```
def build(self):
```

La declaración anterior anula el método de compilación de la clase de aplicación. Lo que devolverá el widget que debe mostrarse cuando inicie la aplicación.

```
return Label(text='Hello world')
```

La declaración anterior es el cuerpo del método de construcción. Está devolviendo la etiqueta con su texto **Hola mundo** .

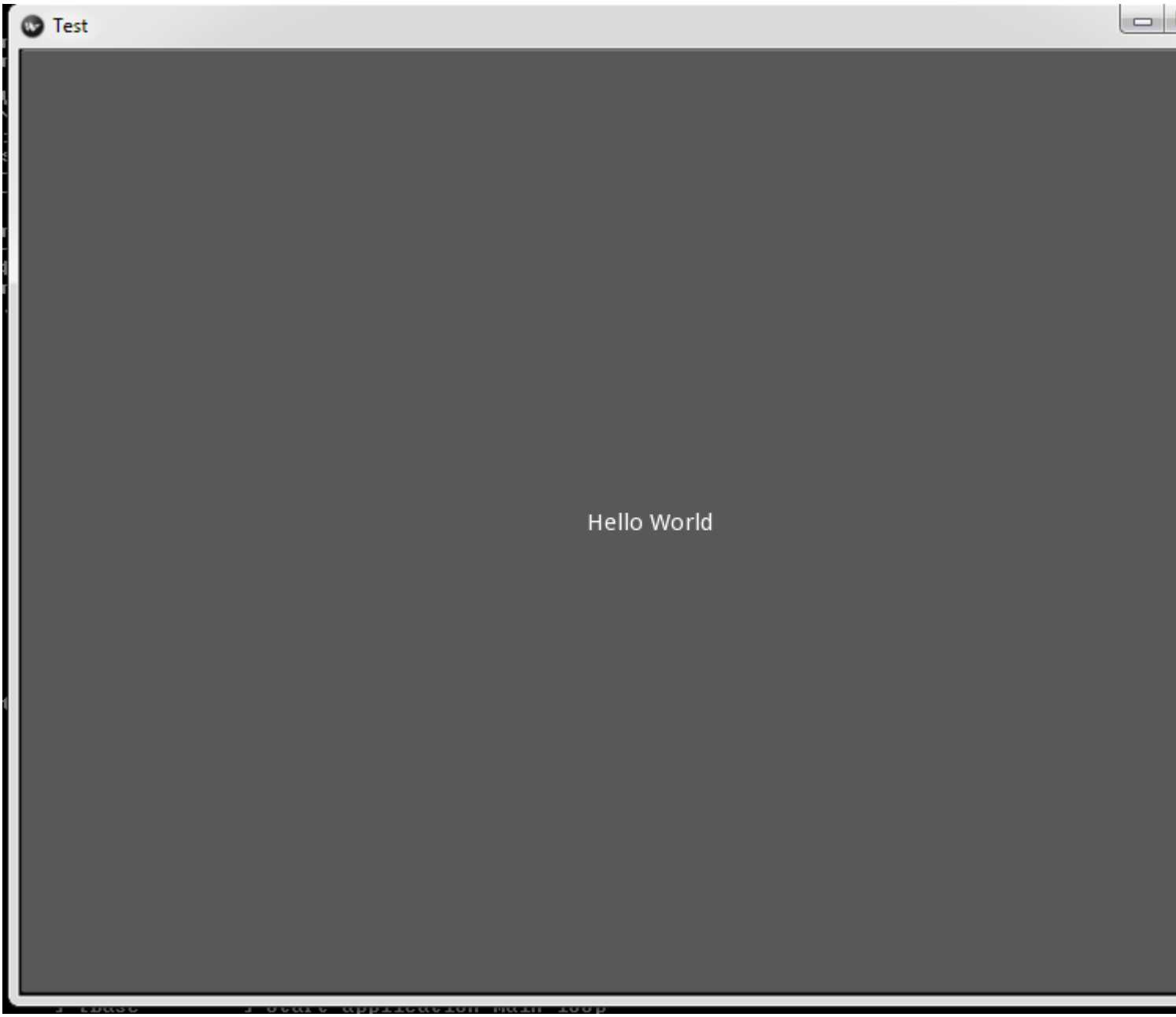
```
if __name__ == '__main__':
```

La declaración anterior es el punto de entrada desde donde el intérprete de Python comienza a ejecutar su aplicación.

```
Test().run()
```

La declaración anterior Inicializa tu clase de prueba creando su instancia. E invoca la función de clase de aplicación run ().

Su aplicación se verá como la imagen de abajo.



Lea kivy - Framework Python multiplataforma para el desarrollo de NUI en línea: https://riptutorial.com/es/python/topic/10743/kivy---framework-python-multiplataforma-para-el-desarrollo-de-nui

Capítulo 106: La declaración de pase

Sintaxis

- pasar

Observaciones

¿Por qué querría decirle al intérprete que no haga nada explícitamente? Python tiene el requisito sintáctico de que los bloques de código (después de, `if` , `except` , `def` , `class` , etc.) no pueden estar vacíos.

Pero a veces un bloque de código vacío es útil en sí mismo. Un bloque de `class` vacío puede definir una nueva clase diferente, como una excepción que se puede capturar. Un bloque de `except` vacío puede ser la forma más simple de expresar "pedir perdón más tarde" si no hubiera nada por lo que pedir perdón. Si un iterador hace todo el trabajo pesado, un iterador vacío `for` ejecutar el iterador puede ser útil.

Por lo tanto, si no se supone que ocurra nada en un bloque de código, se necesita un `pass` para que dicho bloque no produzca un `IndentationError` . Alternativamente, se puede usar cualquier declaración (incluido solo un término para ser evaluado, como el literal de `Ellipsis` `...` o una cadena, más a menudo una cadena de documentación), pero el `pass` deja claro que en realidad no se supone que suceda nada, y no es necesario para ser realmente evaluado y (al menos temporalmente) almacenado en la memoria. Aquí hay una pequeña colección anotada de los usos más frecuentes del `pass` que se cruzó en mi camino, junto con algunos comentarios sobre buenas y malas prácticas.

- Ignorar (todo o) un cierto tipo de `Exception` (ejemplo de `xml`):

```
try:
    self.version = "Expat %d.%d.%d" % expat.version_info
except AttributeError:
    pass # unknown
```

Nota: ignorar todos los tipos de aumentos, como en el siguiente ejemplo de `pandas` , generalmente se considera una mala práctica, ya que también detecta excepciones que probablemente deberían transmitirse a la persona que llama, por ejemplo, `KeyboardInterrupt` o `SystemExit` (o incluso `HardwareIsOnFireError` - ¿Cómo lo sabe?) no se está ejecutando en un cuadro personalizado con errores específicos definidos, que alguna aplicación de llamada querría conocer?).

```
try:
    os.unlink(filename_larry)
except:
    pass
```

En su lugar, usar al menos `except Error:` o en este caso, preferiblemente, `except OSError:` se considera una práctica mucho mejor. Un análisis rápido de todos los módulos de python que he instalado me dio el hecho de que más del 10% de todos, `except ...: pass` declaraciones de `except ...: pass` detectan todas las excepciones, por lo que sigue siendo un patrón frecuente en la programación de python.

- Derivar una clase de excepción que no agrega un comportamiento nuevo (por ejemplo, en `scipy`):

```
class CompileError(Exception):
    pass
```

De manera similar, las clases destinadas a la clase base abstracta a menudo tienen un `__init__` vacío explícito u otros métodos que se supone que derivan las subclases. (por ejemplo, `pebl`)

```
class _BaseSubmittingController(_BaseController):
    def submit(self, tasks): pass
    def retrieve(self, deferred_results): pass
```

- La prueba de que el código se ejecuta correctamente para unos pocos valores de prueba, sin preocuparse por los resultados (de `mpmath`):

```
for x, error in MDNewton(mp, f, (1,-2), verbose=0,
                         norm=lambda x: norm(x, inf)):
    pass
```

- En las definiciones de clase o función, a menudo una cadena de documentación ya está implementada como la *declaración obligatoria* que debe ejecutarse como la única cosa en el bloque. En tales casos, el bloque puede contener un `pass` *además* de la cadena de documentación para decir "Esto no pretende hacer nada", por ejemplo, en `pebl` :

```
class ParsingError(Exception):
    """Error encountered while parsing an ill-formed datafile."""
    pass
```

- En algunos casos, el `pass` se utiliza como marcador de posición para decir "Este método / class / if-block / ... no se ha implementado todavía, pero este será el lugar para hacerlo", aunque personalmente prefiero el literal de `Ellipsis` `...` (NOTA: solo python-3) para diferenciar estrictamente entre esto y el "no-op" intencional en el ejemplo anterior. Por ejemplo, si escribo un modelo a grandes rasgos, podría escribir

```
def update_agent(agent):
    ...
```

donde otros podrían tener

```
def update_agent(agent):
```

```
    pass
```

antes de

```
def time_step(agents):
    for agent in agents:
        update_agent(agent)
```

como un recordatorio para completar la función `update_agent` en un punto posterior, pero ejecute algunas pruebas para ver si el resto del código se comporta como se esperaba. (Una tercera opción para este caso es `raise NotImplementedError` . Esto es útil en particular para dos casos: *“Este método abstracto debe ser implementado por cada subclase, no hay una forma genérica de definirlo en esta clase base”* , o *“Esta función , con este nombre, aún no está implementado en esta versión, pero este es el aspecto que tendrá su firma ”*)

Examples

Ignorar una excepción

```
try:
    metadata = metadata['properties']
except KeyError:
    pass
```

Crear una nueva excepción que puede ser capturada

```
class CompileError(Exception):
    pass
```

Lea La declaración de pase en línea: https://riptutorial.com/es/python/topic/6891/la-declaracion-de-pase

Capítulo 107: La función de impresión

Examples

Fundamentos de impresión

En Python 3 y superior, `print` es una función en lugar de una palabra clave.

```
print('hello world!')
# out: hello world!

foo = 1
bar = 'bar'
baz = 3.14

print(foo)
# out: 1
print(bar)
# out: bar
print(baz)
# out: 3.14
```

También puede pasar una serie de parámetros para `print` :

```
print(foo, bar, baz)
# out: 1 bar 3.14
```

Otra forma de `print` múltiples parámetros es usando un `+`

```
print(str(foo) + " " + bar + " " + str(baz))
# out: 1 bar 3.14
```

Sin embargo, lo que debe tener cuidado al utilizar `+` para imprimir varios parámetros es que el tipo de parámetros debe ser el mismo. Tratar de imprimir el ejemplo anterior sin la conversión de la `string` primero daría como resultado un error, ya que intentaría agregar el número `1` a la cadena `"bar"` y agregarlo al número `3.14` .

```
# Wrong:
# type:int  str  float
print(foo + bar + baz)
# will result in an error
```

Esto se debe a que el contenido de la `print` se evaluará primero:

```
print(4 + 5)
# out: 9
print("4" + "5")
# out: 45
print([4] + [5])
# out: [4, 5]
```

De lo contrario, usar a `+` puede ser muy útil para que un usuario lea la salida de variables. En el siguiente ejemplo, ¡la salida es muy fácil de leer!

El siguiente script demuestra esto

```
import random
#telling python to include a function to create random numbers
randnum = random.randint(0, 12)
#make a random number between 0 and 12 and assign it to a variable
print("The randomly generated number was - " + str(randnum))
```

Puede evitar la `print` la función de imprimir automáticamente una nueva línea utilizando el `end` de parámetros:

```
print("this has no newline at the end of it... ", end="")
print("see?")
# out: this has no newline at the end of it... see?
```

Si desea escribir en un archivo, puede pasarlo como `file` parámetros:

```
with open('my_file.txt', 'w+') as my_file:
    print("this goes to the file!", file=my_file)
```

¡Esto va al archivo!

Parámetros de impresión

Puedes hacer más que solo imprimir texto. `print` también tiene varios parámetros para ayudarte.

Argumento `sep` : coloca una cadena entre argumentos.

¿Necesita imprimir una lista de palabras separadas por una coma o alguna otra cadena?

```
>>> print('apples','bannas', 'cherries', sep=', ')
apple, bannas, cherries
>>> print('apple','banna', 'cherries', sep=', ')
apple, banna, cherries
>>>
```

`end` argumento: use algo que no sea una nueva línea al final

Sin el argumento `end` , todas `print()` funciones `print()` escriben una línea y luego van al principio de la siguiente línea. Puede cambiarlo para que no haga nada (use una cadena vacía de ''), o doble espacio entre párrafos utilizando dos líneas nuevas.

```
>>> print("<a", end=''); print(" class='jidn'" if 1 else "", end=''); print("/>")
<a class='jidn'/>
>>> print("paragraph1", end="\n\n"); print("paragraph2")
paragraph1

paragraph2
```

```
>>>
```

`file` argumento: enviar salida a otro lugar que no sea sys.stdout.

Ahora puede enviar su texto a stdout, a un archivo, oa StringIO y no le importa lo que le den. Si funciona como un archivo, funciona como un archivo.

```
>>> def sendit(out, *values, sep=' ', end='\n'):
...     print(*values, sep=sep, end=end, file=out)
...
>>> sendit(sys.stdout, 'apples', 'bannas', 'cherries', sep='\t')
apples    bannas    cherries
>>> with open("delete-me.txt", "w+") as f:
...    sendit(f, 'apples', 'bannas', 'cherries', sep=' ', end='\n')
...
>>> with open("delete-me.txt", "rt") as f:
...     print(f.read())
...
apples bannas cherries

>>>
```

Hay un cuarto parámetro de `flush` que forzará la descarga de la corriente.

Lea La función de impresión en línea: https://riptutorial.com/es/python/topic/1360/la-funcion-de-impresion

Capítulo 108: La variable especial __name__

Introducción

La variable especial `__name__` se usa para verificar si un archivo ha sido importado como un módulo o no, y para identificar una función, clase, objeto de módulo por su atributo `__name__` .

Observaciones

La variable especial de Python `__name__` se establece en el nombre del módulo que lo contiene. En el nivel superior (como en el intérprete interactivo, o en el archivo principal) se establece en `'__main__'` . Esto se puede usar para ejecutar un bloque de instrucciones si un módulo se ejecuta directamente en lugar de importarse.

El atributo especial relacionado `obj.__name__` se encuentra en las clases, los módulos y las funciones importadas *(incluidos los métodos)* y proporciona el nombre del objeto cuando se define.

Examples

__name__ == '__main__'

La variable especial `__name__` no es establecida por el usuario. Se utiliza principalmente para verificar si el módulo se está ejecutando solo o no porque se realizó una `import` . Para evitar que su módulo ejecute ciertas partes de su código cuando se importa, verifique `if __name__ == '__main__'` .

Deje que **module_1.py** tenga una sola línea de largo:

```
import module2.py
```

Y veamos que pasa, dependiendo de **module2.py.**

Situación 1

módulo2.py

```
print('hello')
```

Ejecutando **module1.py** imprimirá `hello`
Ejecutando **module2.py** imprimirá `hello`

Situación 2

módulo2.py

```
if __name__ == '__main__':
    print('hello')
```

Ejecutando **module1.py** no imprimirá nada
Ejecutando **module2.py** imprimirá `hello`

function_class_or_module .__ name__

El atributo especial `__name__` de una función, clase o módulo es una cadena que contiene su nombre.

```
import os

class C:
    pass

def f(x):
    x += 2
    return x


print(f)
# <function f at 0x029976B0>
print(f.__name__)
# f

print(C)
# <class '__main__.C'>
print(C.__name__)
# C

print(os)
# <module 'os' from '/spam/eggs/'>
print(os.__name__)
# os
```

Sin embargo, el atributo `__name__` no es el nombre de la variable que hace referencia a la clase, el método o la función, sino que es el nombre que se le asigna cuando se define.

```
def f():
    pass

print(f.__name__)
# f - as expected

g = f
print(g.__name__)
# f - even though the variable is named g, the function is still named f
```

Esto se puede utilizar, entre otros, para la depuración:

```
def enter_exit_info(func):
```

```
    def wrapper(*arg, **kw):
        print '-- entering', func.__name__
        res = func(*arg, **kw)
        print '-- exiting', func.__name__
        return res
    return wrapper

@enter_exit_info
def f(x):
    print 'In:', x
    res = x + 2
    print 'Out:', res
    return res

a = f(2)

# Outputs:
#     -- entering f
#     In: 2
#     Out: 4
#     -- exiting f
```

Utilizar en el registro

Al configurar la funcionalidad de `logging` incorporada, un patrón común es crear un registrador con el `__name__` del módulo actual:

```
logger = logging.getLogger(__name__)
```

Esto significa que el nombre completo del módulo aparecerá en los registros, lo que facilitará ver de dónde provienen los mensajes.

Lea La variable especial __name__ en línea: https://riptutorial.com/es/python/topic/1223/la-variable-especial---name--

Capítulo 109: Las clases

Introducción

Python se ofrece a sí mismo no solo como un popular lenguaje de scripting, sino que también es compatible con el paradigma de programación orientado a objetos. Las clases describen datos y proporcionan métodos para manipular esos datos, todos incluidos en un solo objeto. Además, las clases permiten la abstracción al separar los detalles de implementación concretos de las representaciones abstractas de datos.

El código que utiliza clases es generalmente más fácil de leer, entender y mantener.

Examples

Herencia basica

La herencia en Python se basa en ideas similares utilizadas en otros lenguajes orientados a objetos como Java, C ++, etc. Una nueva clase puede derivarse de una clase existente de la siguiente manera.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

`BaseClass` es la clase ya existente (*principal*), y `DerivedClass` es la nueva clase (*secundaria*) que hereda (o *subclases*) atributos de `BaseClass` . **Nota** : a partir de Python 2.2, todas las clases heredan implícitamente de la clase de `object` , que es la clase base para todos los tipos incorporados.

Definimos una clase de `Rectangle` principal en el siguiente ejemplo, que se hereda implícitamente del `object` :

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

La clase `Rectangle` se puede usar como una clase base para definir una clase `Square` , ya que un cuadrado es un caso especial de rectángulo.

```
class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
        super(Square, self).__init__(s, s)
        self.s = s
```

La clase `Square` heredará automáticamente todos los atributos de la clase `Rectangle` , así como la clase de objeto. `super()` se utiliza para llamar al `__init__()` de la clase `Rectangle` , esencialmente llamando a cualquier método anulado de la clase base. **Nota** : en Python 3, `super()` no requiere argumentos.

Los objetos de clase derivados pueden acceder y modificar los atributos de sus clases base:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

Funciones incorporadas que funcionan con herencia.

`issubclass(DerivedClass, BaseClass)` : devuelve `True` si `DerivedClass` es una subclase de `BaseClass`

`isinstance(s, Class)` : devuelve `True` si s es una instancia de `Class` o cualquiera de las clases derivadas de `Class`

```
# subclass check
issubclass(Square, Rectangle)
# Output: True

# instantiate
r = Rectangle(3, 4)
s = Square(2)

isinstance(r, Rectangle)
# Output: True
isinstance(r, Square)
# Output: False
# A rectangle is not a square

isinstance(s, Rectangle)
# Output: True
# A square is a rectangle
isinstance(s, Square)
# Output: True
```

Variables de clase e instancia

Las variables de instancia son únicas para cada instancia, mientras que las variables de clase son compartidas por todas las instancias.

```
class C:
    x = 2  # class variable

    def __init__(self, y):
        self.y = y  # instance variable

C.x
# 2
C.y
# AttributeError: type object 'C' has no attribute 'y'

c1 = C(3)
c1.x
# 2
c1.y
# 3

c2 = C(4)
c2.x
# 2
c2.y
# 4
```

Se puede acceder a las variables de clase en las instancias de esta clase, pero la asignación al atributo de clase creará una variable de instancia que sombrea la variable de clase

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Tenga en cuenta que la *mutación de* variables de clase de instancias puede llevar a algunas consecuencias inesperadas.

```
class D:
    x = []
    def __init__(self, item):
        self.x.append(item)  # note that this is not an assigment!

d1 = D(1)
d2 = D(2)

d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

Métodos enlazados, no enlazados y estáticos.

La idea de métodos enlazados y no enlazados se eliminó en Python 3 . En Python 3 cuando declara un método dentro de una clase, está usando una palabra clave `def` , creando así un objeto de función. Esta es una función regular, y la clase circundante funciona como su espacio de nombres. En el siguiente ejemplo, declaramos el método `f` dentro de la clase `A` , y se convierte en una función `Af` :

Python 3.x 3.0

```
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...>  (in Python 3.x)
```

En Python 2, el comportamiento fue diferente: los objetos de función dentro de la clase fueron reemplazados implícitamente por objetos de tipo `instancemethod` , que se denominaron *métodos* no vinculados porque no estaban vinculados a ninguna instancia de clase en particular. Fue posible acceder a la función subyacente utilizando la propiedad `.__func__` .

Python 2.x 2.3

```
A.f
# <unbound method A.f>   (in Python 2.x)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

Los últimos comportamientos se confirman mediante inspección: los métodos se reconocen como funciones en Python 3, mientras que la distinción se mantiene en Python 2.

Python 3.x 3.0

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Python 2.x 2.3

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

En ambas versiones de Python function / method `Af` se puede llamar directamente, siempre que

pase una instancia de clase `A` como primer argumento.

```
A.f(1, 7)
# Python 2: TypeError: unbound method f() must be called with
#                      A instance as first argument (got int instance instead)
# Python 3: 14
a = A()
A.f(a, 20)
# Python 2 & 3: 40
```

Supongamos ahora que `a` es una instancia de la clase `A` , lo que es `af` entonces? Bueno, intuitivamente este debe ser el mismo método `f` de clase `A` , sólo que debe de alguna manera "saber" que se aplica al objeto `a` - método en Python esto se llama *unida* a `a` .

Los detalles esenciales son los siguientes: writing `af` invoca el método magic `__getattribute__` de `a` , que primero verifica si `a` tiene un atributo llamado `f` (no lo hace), luego verifica la clase `A` si contiene un método con ese nombre (lo hace), y crea un nuevo objeto `m` del `method` de tipo que tiene la referencia al `Af` original en `m.__func__` , y una referencia al objeto `a` en `m.__self__` . Cuando este objeto se llama como una función, simplemente hace lo siguiente: `m(...) => m.__func__(m.__self__, ...)` . Por lo tanto, este objeto se denomina **método enlazado** porque, cuando se invoca, sabe que debe proporcionar el objeto al que estaba vinculado como primer argumento. (Estas cosas funcionan de la misma manera en Python 2 y 3).

```
a = A()
a.f
# <bound method A.f of <__main__.A object at ...>>
a.f(2)
# 4

# Note: the bound method object a.f is recreated *every time* you call it:
a.f is a.f  # False
# As a performance optimization you can store the bound method in the object's
# __dict__, in which case the method object will remain fixed:
a.f = a.f
a.f is a.f  # True
```

Finalmente, Python tiene **métodos de clase** y **métodos estáticos** , tipos especiales de métodos. Los métodos de clase funcionan de la misma manera que los métodos regulares, excepto que cuando se invocan en un objeto, se unen a la *clase* del objeto en lugar de al objeto. Así `m.__self__ = type(a)` . Cuando llama a dicho método enlazado, pasa la clase de `a` como primer argumento. Los métodos estáticos son incluso más simples: no vinculan nada en absoluto, y simplemente devuelven la función subyacente sin ninguna transformación.

```
class D(object):
    multiplier = 2

    @classmethod
    def f(cls, x):
        return cls.multiplier * x

    @staticmethod
    def g(name):
        print("Hello, %s" % name)
```

```
D.f
# <bound method type.f of <class '__main__.D'>>
D.f(12)
# 24
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

Tenga en cuenta que los métodos de clase están vinculados a la clase incluso cuando se accede a ellos en la instancia:

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
# (2, 1337)
d.f
# <bound method D.f of <class '__main__.D'>>
d.f(10)
# 20
```

Vale la pena señalar que en el nivel más bajo, las funciones, los métodos, los métodos estáticos, etc., son en realidad descriptores que invocan los métodos especiales `__get__` , `__set__` y opcionalmente `__del__` . Para más detalles sobre métodos de clase y métodos estáticos:

- ¿Cuál es la diferencia entre @staticmethod y @classmethod en Python?
- ¿Significado de @classmethod y @staticmethod para principiante?

Clases de estilo nuevo vs. estilo antiguo

Python 2.x 2.2.0

Las clases de *nuevo estilo* se introdujeron en Python 2.2 para unificar *clases* y *tipos* . Heredan del tipo de `object` nivel superior. *Una clase de nuevo estilo es un tipo definido por el usuario* y es muy similar a los tipos incorporados.

```
# new-style class
class New(object):
    pass

# new-style instance
new = New()

new.__class__
# <class '__main__.New'>
type(new)
# <class '__main__.New'>
issubclass(New, object)
# True
```

Las clases de *estilo antiguo* **no** heredan de `object` . Las instancias de estilo antiguo siempre se implementan con un tipo de `instance` incorporado.

```
# old-style class
class Old:
    pass

# old-style instance
old = Old()

old.__class__
# <class __main__.Old at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Python 3.x 3.0.0

En Python 3, se eliminaron las clases de estilo antiguo.

Las clases de nuevo estilo en Python 3 heredan implícitamente del `object` , por lo que ya no es necesario especificar `MyClass(object)` .

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

Valores por defecto para variables de instancia

Si la variable contiene un valor de un tipo inmutable (por ejemplo, una cadena), está bien asignar un valor predeterminado como este

```
class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width  * self.height

# Create some instances of the class
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # blue

red_rectangle = Rectangle(2, 3, 'red')
print(red_rectangle.color) # red
```

Hay que tener cuidado al inicializar objetos mutables como listas en el constructor. Considere el siguiente ejemplo:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=[0,0], color='blue'):
        self.width = width
        self.height = height
        self.pos = pos
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [4, 0] r2's pos has changed as well
```

Este comportamiento se debe al hecho de que en Python los parámetros predeterminados están vinculados en la ejecución de la función y no en la declaración de la función. Para obtener una variable de instancia predeterminada que no se comparte entre las instancias, se debe usar una construcción como esta:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # default value is [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [0, 0] r2's pos hasn't changed
```

Consulte también Argumentos predeterminados mutables y “Menos asombro” y el Argumento predeterminado mutable .

Herencia múltiple

Python utiliza el algoritmo de linealización C3 para determinar el orden en el que resolver los atributos de clase, incluidos los métodos. Esto se conoce como Orden de resolución de métodos (MRO).

Aquí hay un ejemplo simple:

```
class Foo(object):
    foo = 'attr foo of Foo'


class Bar(object):
    foo = 'attr foo of Bar' # we won't see this.
    bar = 'attr bar of Bar'

class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'
```

Ahora si creamos una instancia de FooBar, si buscamos el atributo foo, veremos que el atributo

de Foo se encuentra primero

```
fb = FooBar()
```

y

```
>>> fb.foo
'attr foo of Foo'
```

Aquí está el MRO de FooBar:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

Se puede decir simplemente que el algoritmo MRO de Python es

1. Profundidad primero (por ejemplo, `FooBar` luego `Foo`) a menos que
2. un padre compartido (`object`) está bloqueado por un hijo (`Bar`) y
3. No se permiten relaciones circulares.

Es decir, por ejemplo, Bar no puede heredar de FooBar, mientras que FooBar hereda de Bar.

Para un ejemplo completo en Python, vea la entrada de wikipedia .

Otra característica poderosa en la herencia es `super` . super puede obtener características de clases para padres.

```
class Foo(object):
    def foo_method(self):
        print "foo Method"

class Bar(object):
    def bar_method(self):
        print "bar Method"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

Herencia múltiple con el método init de clase, cuando cada clase tiene su propio método init, entonces intentamos obtener una inercia múltiple, luego solo se llama al método init de la clase que se hereda primero.

para el siguiente ejemplo, solo se llama al método de **inicio de** clase Foo que no se llama a la clase de **barra de** inicio

```
    class Foo(object):
        def __init__(self):
            print "foo init"

    class Bar(object):
        def __init__(self):
```

```
            print "bar init"

    class FooBar(Foo, Bar):
        def __init__(self):
            print "foobar init"
            super(FooBar, self).__init__()

    a = FooBar()
```

Salida:

```
    foobar init
    foo init
```

Pero eso no significa que la clase **Bar** no sea hereditaria. La instancia de la clase **FooBar** final también es una instancia de la clase **Bar** y la clase **Foo** .

```
print isinstance(a,FooBar)
print isinstance(a,Foo)
print isinstance(a,Bar)
```

Salida:

```
True
True
True
```

Descriptores y búsquedas punteadas

Los descriptores son objetos que son (generalmente) atributos de clases y que tienen cualquiera de los métodos especiales `__get__` , `__set__` o `__delete__` .

Los descriptores de datos tienen cualquiera de `__set__` , o `__delete__`

Estos pueden controlar la búsqueda de puntos en una instancia y se utilizan para implementar funciones, `staticmethod` , `classmethod` y `property` . Una búsqueda de puntos (por ejemplo, la instancia `foo` de la clase `Foo` busca la `bar` atributos, es decir, `foo.bar`) utiliza el siguiente algoritmo:

1. `bar` se mira en la clase, `Foo` . Si está allí y es un **descriptor de datos** , entonces se usa el descriptor de datos. Así es como la `property` puede controlar el acceso a los datos en una instancia, y las instancias no pueden anular esto. Si un **descriptor de datos** no está allí, entonces

2. `bar` se mira en la instancia `__dict__` . Es por esto que podemos anular o bloquear los métodos que se invocan desde una instancia con una búsqueda de puntos. Si existe una `bar` en la instancia, se utiliza. Si no, entonces nosotros

3. Busca en la clase `Foo` para `bar` . Si es un **Descriptor** , entonces se usa el protocolo del descriptor. Así es como se implementan las funciones (en este contexto, los métodos `classmethod`), el `classmethod` y el `staticmethod` . De lo contrario, simplemente devuelve el

objeto allí, o hay un `AttributeError`

Métodos de clase: inicializadores alternos

Los métodos de clase presentan formas alternativas para construir instancias de clases. Para ilustrar, veamos un ejemplo.

Supongamos que tenemos una clase de `Person` relativamente simple:

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Podría ser útil tener una forma de crear instancias de esta clase especificando un nombre completo en lugar del nombre y apellido por separado. Una forma de hacer esto sería tener el `last_name` como un parámetro opcional, y suponiendo que si no se da, pasamos el nombre completo en:

```
class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Sin embargo, hay dos problemas principales con este bit de código:

1. Los parámetros `first_name` y `last_name` ahora son `last_name` , ya que puede ingresar un nombre completo para `first_name` . Además, si hay más casos y / o más parámetros que tienen este tipo de flexibilidad, la ramificación if / elif / else puede ser molesta rápidamente.

2. No es tan importante, pero aún así vale la pena señalar: ¿y si `last_name` es `None` , pero `first_name` no se divide en dos o más cosas a través de los espacios? Tenemos otra capa de validación de entrada y / o manejo de excepciones ...

Introduzca los métodos de clase. En lugar de tener un solo inicializador, crearemos un inicializador separado, llamado `from_full_name` , y lo `classmethod` decorador de `classmethod` (incorporado).

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    @classmethod
    def from_full_name(cls, name, age):
        if " " not in name:
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        return cls(first_name, last_name, age)

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Note `cls` lugar de `self` como el primer argumento de `from_full_name` . Los métodos de clase se aplican a la clase general, *no a* una instancia de una clase dada (que es lo que el `self` generalmente denota). Por lo tanto, si `cls` es nuestra `Person` de clase, entonces el valor devuelto por la `from_full_name` método de clase es `Person(first_name, last_name, age)` , que utiliza la `Person` 's `__init__` para crear una instancia de la `Person` de clase. En particular, si tuviéramos que hacer una subclase `Employee` of `Person` , `from_full_name` también funcionaría en la clase `Employee` .

Para mostrar que esto funciona como se esperaba, `__init__` instancias de `Person` de más de una manera sin la bifurcación en `__init__` :

```
In [2]: bob = Person("Bob", "Bobberson", 42)

In [3]: alice = Person.from_full_name("Alice Henderson", 31)

In [4]: bob.greet()
Hello, my name is Bob Bobberson.

In [5]: alice.greet()
Hello, my name is Alice Henderson.
```

Otras referencias:

- Python @classmethod y @staticmethod para principiantes?
- https://docs.python.org/2/library/functions.html#classmethod
- https://docs.python.org/3.5/library/functions.html#classmethod

Composición de la clase

La composición de clase permite relaciones explícitas entre objetos. En este ejemplo, las personas viven en ciudades que pertenecen a países. La composición permite a las personas acceder al número de todas las personas que viven en su país:

```
class Country(object):
```

```
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)


class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople


    def addPerson(self, person):
        self.people.append(person)

    def join_country(self,country):
        self.country = country
        country.addCity(self)

        for i in range(self.numPeople):
                person(i).join_city(self)


class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x

US=Country()
NYC=City(10).join_country(US)
SF=City(5).join_country(US)

print(US.cities[0].people[0].people_in_my_country())

# 15
```

Parche de mono

En este caso, "parche de mono" significa agregar una nueva variable o método a una clase después de que se haya definido. Por ejemplo, digamos que definimos la clase `A` como

```
class A(object):
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

Pero ahora queremos agregar otra función más adelante en el código. Supongamos que esta función es la siguiente.

```
def get_num(self):
    return self.num
```

Pero, ¿cómo añadimos esto como un método en `A` ? Eso es simple, simplemente colocamos esa función en `A` con una declaración de asignación.

```
A.get_num = get_num
```

¿Por qué funciona esto? Porque las funciones son objetos como cualquier otro objeto, y los métodos son funciones que pertenecen a la clase.

La función `get_num` estará disponible para todos los existentes (ya creados) así como para las nuevas instancias de `A`

Estas adiciones están disponibles en todas las instancias de esa clase (o sus subclases) automáticamente. Por ejemplo:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42

bar.get_num() # 6
```

Tenga en cuenta que, a diferencia de otros idiomas, esta técnica no funciona para ciertos tipos integrados y no se considera un buen estilo.

Listado de todos los miembros de la clase

La función `dir()` se puede usar para obtener una lista de los miembros de una clase:

```
dir(Class)
```

Por ejemplo:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Es común buscar solo miembros "no mágicos". Esto se puede hacer usando una comprensión simple que enumera los miembros con nombres que no comienzan con __ :

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
```

```
'sort']
```

Advertencias:

Las clases pueden definir un `__dir__()` . Si ese método existe, se llama a `dir()` a `__dir__()` , de lo contrario, Python intentará crear una lista de miembros de la clase. Esto significa que la función dir puede tener resultados inesperados. Dos citas de importancia de la documentación oficial de python :

> Si el objeto no proporciona **dir** (), la función intenta recopilar información del atributo **dict** del objeto, si está definido, y de su tipo de objeto. La lista resultante no está necesariamente completa, y puede ser inexacta cuando el objeto tiene un **getattr** () personalizado.
>
> **Nota:** dado que dir () se proporciona principalmente como una conveniencia para su uso en una solicitud interactiva, intenta proporcionar un conjunto interesante de nombres más de lo que trata de proporcionar un conjunto de nombres definido de manera rigurosa o consistente, y su comportamiento detallado puede cambiar lanzamientos Por ejemplo, los atributos de metaclase no están en la lista de resultados cuando el argumento es una clase.

Introducción a las clases

Una clase, funciona como una plantilla que define las características básicas de un objeto en particular. Aquí hay un ejemplo:

```
class Person(object):
     """A simple class."""                           # docstring
     species = "Homo Sapiens"                        # class attribute

     def __init__(self, name):                       # special method
         """This is the initializer. It's a special
         method (see below).
         """
         self.name = name                            # instance attribute

     def __str__(self):                              # special method
         """This method is run when Python tries
         to cast the object to a string. Return
         this string when using print(), etc.
         """
         return self.name

     def rename(self, renamed):                      # regular method
         """Reassign and print the name attribute."""
         self.name = renamed
         print("Now my name is {}".format(self.name))
```

Hay algunas cosas que se deben tener en cuenta al observar el ejemplo anterior.

1. La clase se compone de *atributos* (datos) y *métodos* (funciones).
2. Los atributos y métodos se definen simplemente como variables y funciones normales.

3. Como se indica en la cadena de documentación correspondiente, el `__init__()` se llama *inicializador* . Es equivalente al constructor en otros lenguajes orientados a objetos, y es el método que se ejecuta por primera vez cuando crea un nuevo objeto o una nueva instancia de la clase.
4. Los atributos que se aplican a toda la clase se definen primero y se denominan *atributos de clase* .
5. Los atributos que se aplican a una instancia específica de una clase (un objeto) se denominan *atributos de instancia* . Generalmente se definen dentro de `__init__()` ; esto no es necesario, pero se recomienda (ya que los atributos definidos fuera de `__init__()` corren el riesgo de ser accedidos antes de que se definan).
6. Cada método, incluido en la definición de clase, pasa el objeto en cuestión como su primer parámetro. La palabra `self` se usa para este parámetro (el uso de `self` es en realidad por convención, ya que la palabra `self` no tiene un significado inherente en Python, pero esta es una de las convenciones más respetadas de Python, y siempre se debe seguir).
7. Aquellos que están acostumbrados a la programación orientada a objetos en otros lenguajes pueden sorprenderse por algunas cosas. Una es que Python no tiene un concepto real de elementos `private` , por lo que todo, de manera predeterminada, imita el comportamiento de la palabra clave `public` C ++ / Java. Para obtener más información, consulte el ejemplo "Miembros de la clase privada" en esta página.
8. Algunos de los métodos de la clase tienen la siguiente forma: `__functionname__(self, other_stuff)` . Todos estos métodos se denominan "métodos mágicos" y son una parte importante de las clases en Python. Por ejemplo, la sobrecarga de operadores en Python se implementa con métodos mágicos. Para más información, consulte la documentación relevante .

Ahora vamos a hacer algunos ejemplos de nuestra clase de `Person` !

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

Actualmente tenemos tres objetos `Person` , `kelly` , `joseph` y `john_doe` .

Podemos acceder a los atributos de la clase desde cada instancia utilizando el operador de punto `.` Note nuevamente la diferencia entre los atributos de clase e instancia:

```
>>> # Attributes
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

Podemos ejecutar los métodos de la clase usando el mismo operador de punto `.` :

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>>  print(john_doe)
'John Doe'
>>>  john_doe.rename("John")
'Now my name is John'
```

Propiedades

Las clases de Python admiten **propiedades** , que parecen variables de objetos normales, pero con la posibilidad de adjuntar comportamiento y documentación personalizados.

```
class MyClass(object):

    def __init__(self):
       self._my_string = ""

    @property
    def string(self):
        """A profoundly important string."""
        return self._my_string

    @string.setter
    def string(self, new_value):
        assert isinstance(new_value, str), \
               "Give me a string, not a %r!" % type(new_value)
        self._my_string = new_value

    @string.deleter
    def x(self):
        self._my_string = None
```

De la clase de objeto `MyClass` *parecerá* tener tiene una propiedad `.string` , sin embargo su comportamiento está ahora estrictamente controlado:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

Además de la sintaxis útil que se mencionó anteriormente, la sintaxis de la propiedad permite la validación u otros aumentos a esos atributos. Esto podría ser especialmente útil con las API públicas, donde se debe brindar un nivel de ayuda al usuario.

Otro uso común de las propiedades es permitir que la clase presente "atributos virtuales", atributos que no se almacenan en realidad, sino que se calculan solo cuando se solicitan.

```
class Character(object):
    def __init__(name, max_hp):
        self._name = name
        self._hp = max_hp
        self._max_hp = max_hp
```

```
    # Make hp read only by not providing a set method
    @property
    def hp(self):
        return self._hp

    # Make name read only by not providing a set method
    @property
    def name(self):
        return self.name

    def take_damage(self, damage):
        self.hp -= damage
        self.hp = 0 if self.hp <0 else self.hp

    @property
    def is_alive(self):
        return self.hp != 0

    @property
    def is_wounded(self):
        return self.hp < self.max_hp if self.hp > 0 else False

    @property
    def is_dead(self):
        return not self.is_alive

bilbo = Character('Bilbo Baggins', 100)
bilbo.hp
# out : 100
bilbo.hp = 200
# out : AttributeError: can't set attribute
# hp attribute is read only.

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )

bilbo.hp
# out : 50

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : True
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )
bilbo.hp
# out : 0

bilbo.is_alive
# out : False
bilbo.is_wounded
# out : False
bilbo.is_dead
```

```
# out : True
```

Clase de singleton

Un singleton es un patrón que restringe la creación de instancias de una clase a una instancia / objeto. Para más información sobre los patrones de diseño singleton de python, consulte aquí .

```
class Singleton:
    def __new__(cls):
        try:
            it = cls.__it__
        except AttributeError:
            it = cls.__it__ = object.__new__(cls)
        return it

    def __repr__(self):
        return '<{}>'.format(self.__class__.__name__.upper())

    def __eq__(self, other):
         return other is self
```

Otro método es decorar tu clase. Siguiendo el ejemplo de esta respuesta, crea una clase Singleton:

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.

    The decorated class can define one `__init__` function that
    takes only the `self` argument. Other than that, there are
    no restrictions that apply to the decorated class.

    To get the singleton instance, use the `Instance` method. Trying
    to use `__call__` will result in a `TypeError` being raised.

    Limitations: The decorated class cannot be inherited from.

    """

    def __init__(self, decorated):
        self._decorated = decorated

    def Instance(self):
        """
        Returns the singleton instance. Upon its first call, it creates a
        new instance of the decorated class and calls its `__init__` method.
        On all subsequent calls, the already created instance is returned.

        """
        try:
            return self._instance
        except AttributeError:
            self._instance = self._decorated()
            return self._instance
```

```
    def __call__(self):
        raise TypeError('Singletons must be accessed through `Instance()`.')

    def __instancecheck__(self, inst):
        return isinstance(inst, self._decorated)
```

Para usar puedes usar el método de `Instance`

```
@Singleton
class Single:
    def __init__(self):
        self.name=None
        self.val=0
    def getName(self):
        print(self.name)

x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # outputs I'm single
y.getName() # outputs I'm single
```

Lea Las clases en línea: https://riptutorial.com/es/python/topic/419/las-clases

Capítulo 110: Lectura y Escritura CSV

Examples

Escribiendo un archivo TSV

Pitón

```
import csv

with open('/tmp/output.tsv', 'wt') as out_file:
    tsv_writer = csv.writer(out_file, delimiter='\t')
    tsv_writer.writerow(['name', 'field'])
    tsv_writer.writerow(['Dijkstra', 'Computer Science'])
    tsv_writer.writerow(['Shelah', 'Math'])
    tsv_writer.writerow(['Aumann', 'Economic Sciences'])
```

Archivo de salida

```
$ cat /tmp/output.tsv

name    field
Dijkstra    Computer Science
Shelah    Math
Aumann    Economic Sciences
```

Usando pandas

Escriba un archivo CSV desde un `dict` o un `DataFrame` .

```
import pandas as pd

d = {'a': (1, 101), 'b': (2, 202), 'c': (3, 303)}
pd.DataFrame.from_dict(d, orient="index")
df.to_csv("data.csv")
```

Lea un archivo CSV como un `DataFrame` y `DataFrame` a un `dict` :

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

Lea Lectura y Escritura CSV en línea: https://riptutorial.com/es/python/topic/2116/lectura-y-escritura-csv

Capítulo 111: Lista

Introducción

La **Lista de** Python es una estructura de datos general ampliamente utilizada en los programas de Python. Se encuentran en otros idiomas, a menudo denominados *arreglos dinámicos* . Ambos son *mutables* y un tipo de datos de *secuencia* que les permite ser *indexados* y *segmentados* . La lista puede contener diferentes tipos de objetos, incluidos otros objetos de lista.

Sintaxis

- [valor, valor, ...]
- lista ([iterable])

Observaciones

`list` es un tipo particular de iterable, pero no es el único que existe en Python. A veces será mejor usar `set` , `tuple` o `dictionary`

`list` es el nombre dado en Python a arreglos dinámicos (similar al `vector<void*>` de C ++ o `ArrayList<Object>` de Java `ArrayList<Object>`). No es una lista enlazada.

El acceso a los elementos se realiza en tiempo constante y es muy rápido. Anexar elementos al final de la lista es tiempo constante amortizado, pero de vez en cuando puede implicar la asignación y copia de toda la `list` .

Las comprensiones de listas están relacionadas con listas.

Examples

Acceso a los valores de la lista

Las listas de Python están indexadas a cero, y actúan como matrices en otros idiomas.

```
lst = [1, 2, 3, 4]
lst[0]  # 1
lst[1]  # 2
```

Intentar acceder a un índice fuera de los límites de la lista generará un `IndexError` .

```
lst[4]  # IndexError: list index out of range
```

Los índices negativos se interpretan como contadores desde el *final* de la lista.

```
lst[-1]  # 4
```

```
lst[-2]  # 3
lst[-5]  # IndexError: list index out of range
```

Esto es funcionalmente equivalente a

```
lst[len(lst)-1]  # 4
```

Las listas permiten usar *notación de* `lst[start:end:step]` como `lst[start:end:step]` . La salida de la notación de división es una nueva lista que contiene elementos desde el `start` hasta el `end-1` del índice. Si se omiten las opciones, `start` defecto al principio de la lista, de `end` a extremo de la lista y `step` al 1:

```
lst[1:]      # [2, 3, 4]
lst[:3]      # [1, 2, 3]
lst[::2]     # [1, 3]
lst[::-1]    # [4, 3, 2, 1]
lst[-1:0:-1] # [4, 3, 2]
lst[5:8]     # [] since starting index is greater than length of lst, returns empty list
lst[1:10]    # [2, 3, 4] same as omitting ending index
```

Con esto en mente, puede imprimir una versión invertida de la lista llamando

```
lst[::-1]    # [4, 3, 2, 1]
```

Cuando se usan longitudes de pasos de cantidades negativas, el índice inicial debe ser mayor que el índice final, de lo contrario, el resultado será una lista vacía.

```
lst[3:1:-1] # [4, 3]
```

El uso de índices de pasos negativos es equivalente al siguiente código:

```
reversed(lst)[0:2] # 0 = 1 -1
                   # 2 = 3 -1
```

Los índices utilizados son 1 menos que los utilizados en la indexación negativa y se invierten.

Rebanado avanzado

Cuando las listas se `__getitem__()` se llama al método `__getitem__()` del objeto de lista, con un objeto de `slice` . Python tiene un método de división integrado para generar objetos de división. Podemos usar esto para *almacenar* una porción y reutilizarla más tarde como así,

```
data = 'chandan purohit    22 2000'  #assuming data fields of fixed length
name_slice = slice(0,19)
age_slice = slice(19,21)
salary_slice = slice(22,None)

#now we can have more readable slices
print(data[name_slice]) #chandan purohit
print(data[age_slice]) #'22'
print(data[salary_slice]) #'2000'
```

Esto puede ser de gran utilidad al proporcionar funcionalidad de corte a nuestros objetos al reemplazar `__getitem__` en nuestra clase.

Lista de métodos y operadores soportados.

Comenzando con una lista dada `a` :

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` : agrega un nuevo elemento al final de la lista.

```
# Append values 6, 7, and 7 to the list
a.append(6)
a.append(7)
a.append(7)
# a: [1, 2, 3, 4, 5, 6, 7, 7]

# Append another list
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]

# Append an element of a different type, as list elements do not need to have the same
type
my_string = "hello world"
a.append(my_string)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
```

   Tenga en cuenta que el método `append()` solo agrega un elemento nuevo al final de la lista. Si agrega una lista a otra, la lista que agregue se convertirá en un elemento único al final de la primera lista.

```
# Appending a list to another list
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
a[8]
# Returns: [8,9]
```

2. `extend(enumerable)` : extiende la lista agregando elementos de otro enumerable.

```
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9, 10]

# Extend list by appending all elements from b
a.extend(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]

# Extend list with elements from a non-list enumerable:
a.extend(range(3))
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
```

   Las listas también se pueden concatenar con el operador `+` . Tenga en cuenta que esto no

modifica ninguna de las listas originales:

```
a = [1, 2, 3, 4, 5, 6] + [7, 7] + b
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

3. `index(value, [startIndex])` índice de inicio `index(value, [startIndex])` : obtiene el índice de la primera aparición del valor de entrada. Si el valor de entrada no está en la lista, se `ValueError` una excepción `ValueError` . Si se proporciona un segundo argumento, la búsqueda se inicia en ese índice especificado.

```
a.index(7)
# Returns: 6

a.index(49) # ValueError, because 49 is not in a.

a.index(7, 7)
# Returns: 7

a.index(7, 8) # ValueError, because there is no 7 starting at index 8
```

4. `insert(index, value)` : inserta un `value` justo antes del `index` especificado. Así después de la inserción el nuevo elemento ocupa el `index` posición.

```
a.insert(0, 0)  # insert 0 at position 0
a.insert(2, 5)  # insert 5 at position 2
# a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

5. `pop([index])` : elimina y devuelve el elemento en el `index` . Sin ningún argumento, elimina y devuelve el último elemento de la lista.

```
a.pop(2)
# Returns: 5
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
a.pop(8)
# Returns: 7
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# With no argument:
a.pop()
# Returns: 10
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. `remove(value)` : elimina la primera aparición del valor especificado. Si no se puede encontrar el valor proporcionado, se `ValueError` un `ValueError` .

```
a.remove(0)
a.remove(9)
# a: [1, 2, 3, 4, 5, 6, 7, 8]
a.remove(10)
# ValueError, because 10 is not in a
```

7. `reverse()` : invierte la lista en el lugar y devuelve `None` .

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

También hay otras formas de revertir una lista .

8. `count(value)` : cuenta el número de apariciones de algún valor en la lista.

```
a.count(7)
# Returns: 2
```

9. `sort()` : ordena la lista en orden numérico y lexicográfico y devuelve `None` .

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

Las listas también se pueden revertir cuando se ordenan usando la `reverse=True` en el método `sort()` .

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

Si desea ordenar por atributos de elementos, puede usar el argumento de palabra `key` clave:

```
import datetime

class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height

    def __repr__(self):
        return self.name

l = [Person("John Cena", datetime.date(1992, 9, 12), 175),
     Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
     Person("Jon Skeet", datetime.date(1991, 7, 6), 185)]

l.sort(key=lambda item: item.name)
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item.birthday)
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item.height)
# l: [John Cena, Chuck Norris, Jon Skeet]
```

En caso de lista de dictados el concepto es el mismo:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'height': 175},
```

```
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'height': 180},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'height': 185}]

l.sort(key=lambda item: item['name'])
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item['birthday'])
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Ordenar por subdivisión:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'size': {'height': 175,
'weight': 100}},
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'size' : {'height': 180,
'weight': 90}},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'size': {'height': 185,
'weight': 110}}]

l.sort(key=lambda item: item['size']['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Mejor manera de ordenar usando `attrgetter` y `itemgetter`

Las listas también se pueden clasificar utilizando las funciones `attrgetter` y `itemgetter` del módulo del operador. Estos pueden ayudar a mejorar la legibilidad y la reutilización. Aquí hay unos ejemplos,

```
from operator import itemgetter,attrgetter

people = [{'name':'chandan','age':20,'salary':2000},
          {'name':'chetan','age':18,'salary':5000},
          {'name':'guru','age':30,'salary':3000}]
by_age = itemgetter('age')
by_salary = itemgetter('salary')

people.sort(key=by_age) #in-place sorting by age
people.sort(key=by_salary) #in-place sorting by salary
```

`itemgetter` también se puede dar un índice. Esto es útil si desea ordenar según los índices de una tupla.

```
list_of_tuples = [(1,2), (3,4), (5,0)]
list_of_tuples.sort(key=itemgetter(1))
print(list_of_tuples) #[(5, 0), (1, 2), (3, 4)]
```

Utilice el `attrgetter` si desea ordenar por atributos de un objeto,

```
persons = [Person("John Cena", datetime.date(1992, 9, 12), 175),
           Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
```

```
          Person("Jon Skeet", datetime.date(1991, 7, 6), 185)] #reusing Person class from
above example

person.sort(key=attrgetter('name')) #sort by name
by_birthday = attrgetter('birthday')
person.sort(key=by_birthday) #sort by birthday
```

10. `clear()` : elimina todos los elementos de la lista

    ```
    a.clear()
    # a = []
    ```

11. **Replicación** : multiplicar una lista existente por un número entero producirá una lista más grande que consiste en tantas copias del original. Esto puede ser útil, por ejemplo, para la inicialización de listas:

    ```
    b = ["blah"] * 3
    # b = ["blah", "blah", "blah"]
    b = [1, 3, 5] * 5
    # [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
    ```

    Tenga cuidado al hacer esto si su lista contiene referencias a objetos (por ejemplo, una lista de listas), vea Errores comunes: multiplicación de listas y referencias comunes .

12. **Eliminación de elementos** : es posible eliminar varios elementos de la lista utilizando la palabra clave `del` y la notación de segmento:

    ```
    a = list(range(10))
    del a[::2]
    # a = [1, 3, 5, 7, 9]
    del a[-1]
    # a = [1, 3, 5, 7]
    del a[:]
    # a = []
    ```

13. **Proceso de copiar**

    La asignación predeterminada "=" asigna una referencia de la lista original al nuevo nombre. Es decir, el nombre original y el nuevo nombre apuntan al mismo objeto de lista. Los cambios realizados a través de cualquiera de ellos se reflejarán en otro. Esto a menudo no es lo que pretendías.

    ```
    b = a
    a.append(6)
    # b: [1, 2, 3, 4, 5, 6]
    ```

    Si desea crear una copia de la lista, tiene las siguientes opciones.

    Puedes cortarlo:

    ```
    new_list = old_list[:]
    ```

Puedes usar la función integrada en la lista ():

```
new_list = list(old_list)
```

Puedes usar copy.copy genérico ():

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

Esto es un poco más lento que list () porque primero tiene que averiguar el tipo de datos de old_list.

Si la lista contiene objetos y también desea copiarlos, use copy.deepcopy genérico ():

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Obviamente, el método más lento y que más memoria necesita, pero a veces inevitable.

Python 3.x 3.0

`copy()` - Devuelve una copia superficial de la lista

```
    aa = a.copy()
    # aa = [1, 2, 3, 4, 5]
```

Longitud de una lista

Use `len()` para obtener la longitud unidimensional de una lista.

```
len(['one', 'two'])  # returns 2

len(['one', [2, 3], 'four'])  # returns 3, not 4
```

`len()` también funciona en cadenas, diccionarios y otras estructuras de datos similares a las listas.

Tenga en cuenta que `len()` es una función incorporada, no un método de un objeto de lista.

También tenga en cuenta que el costo de `len()` es `O(1)` , lo que significa que tomará la misma cantidad de tiempo para obtener la longitud de una lista, independientemente de su longitud.

Iterando sobre una lista

Python admite el uso de un bucle `for` directamente en una lista:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)

# Output: foo
```

```
# Output: bar
# Output: baz
```

También puede obtener la posición de cada elemento al mismo tiempo:

```
for (index, item) in enumerate(my_list):
    print('The item in position {} is: {}'.format(index, item))

# Output: The item in position 0 is: foo
# Output: The item in position 1 is: bar
# Output: The item in position 2 is: baz
```

La otra forma de iterar una lista basada en el valor del índice:

```
for i in range(0,len(my_list)):
    print(my_list[i])
#output:
>>>
foo
bar
baz
```

Tenga en cuenta que cambiar elementos en una lista mientras se iteran en ella puede tener resultados inesperados:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)

# Output: foo
# Output: baz
```

En este último ejemplo, eliminamos el primer elemento en la primera iteración, pero eso hizo que se omitiera la `bar` .

Comprobando si un artículo está en una lista

Python hace que sea muy sencillo comprobar si un elemento está en una lista. Simplemente use el operador `in` .

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True

'toast' in lst
# Out: False
```

> Nota: el operador `in` en conjuntos es asintóticamente más rápido que en las listas. Si necesita usarlo muchas veces en listas potencialmente grandes, puede convertir su `list` en un `set` y probar la presencia de elementos en el `set` .

```
slst = set(lst)
'test' in slst
# Out: True
```

Elementos de la lista de inversión

Puede utilizar la función `reversed` que devuelve un iterador a la lista invertida:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Tenga en cuenta que la lista de "números" permanece sin cambios por esta operación, y permanece en el mismo orden en que estaba originalmente.

Para invertir en su lugar, también puede utilizar el método `reverse` .

También puede revertir una lista (al obtener una copia, la lista original no se ve afectada) utilizando la sintaxis de corte, estableciendo el tercer argumento (el paso) como -1:

```
In [1]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: numbers[::-1]
Out[2]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Comprobando si la lista está vacía

El vacío de una lista está asociado al booleano `False` , por lo que no tiene que marcar `len(lst) == 0` , sino solo `lst` o `not lst`

```
lst = []
if not lst:
    print("list is empty")

# Output: list is empty
```

Concatenar y fusionar listas

1. **La forma más sencilla de concatenar** `list1` y `list2` :

   ```
   merged = list1 + list2
   ```

2. **`zip` devuelve una lista de tuplas** , donde la i-th tupla contiene el elemento i-th de cada una de las secuencias de argumentos o iterables:

   ```
   alist = ['a1', 'a2', 'a3']
   blist = ['b1', 'b2', 'b3']

   for a, b in zip(alist, blist):
   ```

```
    print(a, b)

# Output:
# a1 b1
# a2 b2
# a3 b3
```

Si las listas tienen diferentes longitudes, el resultado incluirá solo tantos elementos como el más corto:

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3', 'b4']
for a, b in zip(alist, blist):
    print(a, b)

# Output:
# a1 b1
# a2 b2
# a3 b3

alist = []
len(list(zip(alist, blist)))

# Output:
# 0
```

Para listas de relleno de longitud desigual a la más larga con `None` s use `itertools.zip_longest` (`itertools.izip_longest` en Python 2)

```
alist = ['a1', 'a2', 'a3']
blist = ['b1']
clist = ['c1', 'c2', 'c3', 'c4']

for a,b,c in itertools.zip_longest(alist, blist, clist):
    print(a, b, c)

# Output:
# a1 b1 c1
# a2 None c2
# a3 None c3
# None None c4
```

3. **Insertar en un índice de valores específicos:**

```
 alist = [123, 'xyz', 'zara', 'abc']
 alist.insert(3, [2009])
 print("Final List :", alist)
```

Salida:

```
 Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

Todos y cada uno

Puede usar `all()` para determinar si todos los valores en una evaluación iterable a True

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

Del mismo modo, `any()` determina si uno o más valores en una evaluación iterable a True

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

Si bien este ejemplo utiliza una lista, es importante tener en cuenta que estos elementos integrados funcionan con cualquier iterable, incluidos los generadores.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

Eliminar valores duplicados en la lista

La eliminación de valores duplicados en una lista se puede hacer convirtiendo la lista en un `set` (es decir, una colección desordenada de objetos distintos). Si se necesita una estructura de datos de `list` , entonces el conjunto se puede convertir de nuevo a una lista usando la `list()` funciones `list()` :

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

Tenga en cuenta que al convertir una lista en un conjunto, el pedido original se pierde.

Para preservar el orden de la lista, se puede usar un `OrderedDict`

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

Acceso a valores en lista anidada

Comenzando con una lista tridimensional:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

Accediendo a los elementos de la lista:

```
print(alist[0][0][1])
#2
#Accesses second element in the first list in the first list

print(alist[1][1][2])
#10
#Accesses the third element in the second list in the second list
```

Realizando operaciones de soporte:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Appends 11 to the end of the first list in the first list
```

Usando anidados para bucles para imprimir la lista:

```
for row in alist: #One way to loop through nested lists
    for col in row:
        print(col)
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#[12, 13, 14]
```

Tenga en cuenta que esta operación se puede utilizar en una lista de comprensión o incluso como un generador para producir eficiencias, por ejemplo:

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

No todos los elementos en las listas externas tienen que ser listas ellos mismos:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Otra forma de utilizar anidados para bucles. La otra forma es mejor, pero he necesitado usar esto en ocasiones:

```
for row in range(len(alist)): #A less Pythonic way to loop through lists
    for col in range(len(alist[row])):
       print(alist[row][col])

#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#15
#[12, 13, 14]
```

Usando rebanadas en la lista anidada:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

La lista final:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

Comparacion de listas

Es posible comparar listas y otras secuencias lexicográficamente usando operadores de comparación. Ambos operandos deben ser del mismo tipo.

```
[1, 10, 100] < [2, 10, 100]
# True, because 1 < 2
[1, 10, 100] < [1, 10, 100]
# False, because the lists are equal
[1, 10, 100] <= [1, 10, 100]
# True, because the lists are equal
[1, 10, 100] < [1, 10, 101]
# True, because 100 < 101
[1, 10, 100] < [0, 10, 100]
# False, because 0 < 1
```

Si una de las listas está contenida al comienzo de la otra, la lista más corta gana.

```
[1, 10] < [1, 10, 100]
# True
```

Inicializando una lista a un número fijo de elementos

Para elementos **inmutables** (por ejemplo, `None` , literales de cadenas, etc.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

Para elementos **mutables** , la misma construcción dará como resultado que todos los elementos de la lista se refieran al mismo objeto, por ejemplo, para un conjunto:

```
>>> my_list=[{1}] * 10
>>> print(my_list)
[{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}]
>>> my_list[0].add(2)
>>> print(my_list)
[{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}]
```

En su lugar, para inicializar la lista con un número fijo de objetos **mutables diferentes** , use:

```
my_list=[{1} for _ in range(10)]
```

Lea Lista en línea: https://riptutorial.com/es/python/topic/209/lista

Capítulo 112: Lista de Comprensiones

Introducción

Una lista de comprensión es una herramienta sintáctica para crear listas de forma natural y concisa, como se ilustra en el siguiente código para hacer una lista de cuadrados de los números del 1 al 10: `[i ** 2 for i in range(1,11)]` El dummy `i` de un `range` lista existente se usa para hacer un nuevo patrón de elemento. Se usa donde un bucle for sería necesario en lenguajes menos expresivos.

Sintaxis

- [i for i in range (10)] # lista de comprensión básica
- [i for i in xrange (10)] # lista de comprensión básica con objeto generador en Python 2.x
- [i for i in range (20) if i% 2 == 0] # con filtro
- [x + y para x en [1, 2, 3] para y en [3, 4, 5]] # bucles anidados
- [i if i> 6 else 0 for i in range (10)] # expresión ternaria
- [i if i> 4 else 0 para i en el rango (20) if i% 2 == 0] # con filtro y expresión ternaria
- [[x + y para x en [1, 2, 3]] para y en [3, 4, 5]] # comprensión de lista anidada

Observaciones

La lista de comprensión se describió en PEP 202 y se introdujo en Python 2.0.

Examples

Lista de comprensiones condicionales

Dada una lista de comprensión , puede agregar uno o más `if` condiciones para filtrar los valores.

```
[<expression> for <element> in <iterable> if <condition>]
```

Para cada `<element>` en `<iterable>` ; Si `<condition>` evalúa como `True` , agregue `<expression>` (generalmente una función de `<element>`) a la lista devuelta.

Por ejemplo, esto se puede usar para extraer solo números pares de una secuencia de enteros:

```
[x for x in range(10) if x % 2 == 0]
# Out: [0, 2, 4, 6, 8]
```

Demo en vivo

El código anterior es equivalente a:

```
even_numbers = []
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# Out: [0, 2, 4, 6, 8]
```

Además, una comprensión condicional de la lista de la forma `[e for x in y if c]` (donde `e` y `c` son expresiones en términos de `x`) es equivalente a `list(filter(lambda x: c, map(lambda x: e, y)))` .

A pesar de proporcionar el mismo resultado, preste atención al hecho de que el ejemplo anterior es casi 2 veces más rápido que el segundo. Para aquellos que tienen curiosidad, esta es una buena explicación de la razón.

Tenga en cuenta que esto es bastante diferente de la expresión condicional `... if ... else ...` (a veces conocida como expresión ternaria) que puede usar para la parte `<expression>` de la lista de comprensión. Considere el siguiente ejemplo:

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

Demo en vivo

Aquí, la expresión condicional no es un filtro, sino un operador que determina el valor que se utilizará para los elementos de la lista:

```
<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>
```

Esto se vuelve más obvio si lo combinas con otros operadores:

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in range(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Demo en vivo

Si está utilizando Python 2.7, `xrange` puede ser mejor que el `range` por varios motivos, como se describe en la documentación de `xrange` .

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in xrange(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

El código anterior es equivalente a:

```
numbers = []
for x in range(10):
    if x % 2 == 0:
        temp = x
    else:
        temp = -1
```

```
    numbers.append(2 * temp + 1)
print(numbers)
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Se puede combinar expresiones ternarios y `if` las condiciones. El operador ternario trabaja en el resultado filtrado:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

Lo mismo no podría haberse logrado solo por el operador ternario:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out:['*', '*', '*', '*', 4, '*', 6, '*', 8, '*']
```

Vea también: Filtros , que a menudo proporcionan una alternativa suficiente a las comprensiones de listas condicionales.

Lista de Comprensiones con Bucles Anidados

Las Comprensiones de lista pueden usar anidados `for` bucles. Puede codificar cualquier número de bucles for anidados dentro de una lista por comprensión, y cada uno `for` bucle puede tener una opción asociada `if` la prueba. Al hacerlo, el orden de la `for` las construcciones es del mismo orden que la hora de escribir una serie de anidado `for` declaraciones. La estructura general de las listas de comprensión se ve así:

```
[ expression for target1 in iterable1 [if condition1]
             for target2 in iterable2 [if condition2]...
             for targetN in iterableN [if conditionN] ]
```

Por ejemplo, el siguiente código aplanamiento una lista de listas utilizando múltiples `for` declaraciones:

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

se puede escribir de forma equivalente como una lista de comprensión con múltiples `for` construcciones:

```
data = [[1, 2], [3, 4], [5, 6]]
output = [element for each_list in data for element in each_list]
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

Demo en vivo

Tanto en la forma expandida como en la lista de comprensión, el bucle externo (primero para la declaración) aparece primero.

Además de ser más compacto, la comprensión anidada también es significativamente más rápida.

```
In [1]: data = [[1,2],[3,4],[5,6]]
In [2]: def f():
   ...:     output=[]
   ...:     for each_list in data:
   ...:         for element in each_list:
   ...:             output.append(element)
   ...:     return output
In [3]: timeit f()
1000000 loops, best of 3: 1.37 µs per loop
In [4]: timeit [inner for outer in data for inner in outer]
1000000 loops, best of 3: 632 ns per loop
```

La sobrecarga para la llamada de función anterior es de aproximadamente *140 ns* .

En línea `if` s están anidados de manera similar, y puede ocurrir en cualquier posición después de la primera `for` :

```
data = [[1], [2, 3], [4, 5]]
output = [element for each_list in data
                if len(each_list) == 2
                for element in each_list
                if element != 5]
print(output)
# Out: [2, 3, 4]
```

Demo en vivo

Sin embargo, por razones de legibilidad, debe considerar el uso *de bucles for* tradicionales. Esto es especialmente cierto cuando el anidamiento tiene más de 2 niveles de profundidad y / o la lógica de la comprensión es demasiado compleja. la comprensión de múltiples listas de bucles anidadas podría ser propensa a errores o dar un resultado inesperado.

Refactorización de filtro y mapa para enumerar las comprensiones.

Las funciones de `filter` o `map` menudo deben ser reemplazadas por listas de comprensión . Guido Van Rossum describe esto bien en una carta abierta en 2005 :

> `filter(P, S)` casi siempre se escribe más claro como `[x for x in S if P(x)]` , y esto tiene la gran ventaja de que los usos más comunes incluyen predicados que son comparaciones, por ejemplo, `x==42` , y la definición de un lambda para eso solo requiere mucho más esfuerzo para el lector (más el lambda es más lento que la lista de comprensión). Más aún para el `map(F, S)` que se convierte en `[F(x) for x in S]` . Por supuesto, en muchos casos podrías usar expresiones generadoras.

Las siguientes líneas de código se consideran " *no pythonic* " y generarán errores en muchos linters de python.

```
filter(lambda x: x % 2 == 0, range(10)) # even numbers < 10
map(lambda x: 2*x, range(10)) # multiply each number by two
reduce(lambda x,y: x+y, range(10)) # sum of all elements in list
```

Tomando lo que hemos aprendido de la cita anterior, podemos desglosar estas expresiones de `filter` y `map` en sus *listas de comprensión* equivalentes; También elimina las funciones *lambda* de cada una, lo que hace que el código sea más legible en el proceso.

```
# Filter:
# P(x) = x % 2 == 0
# S = range(10)
[x for x in range(10) if x % 2 == 0]

# Map
# F(x) = 2*x
# S = range(10)
[2*x for x in range(10)]
```

La legibilidad se vuelve aún más evidente cuando se trata de funciones de encadenamiento. Donde debido a la legibilidad, los resultados de un mapa o función de filtro deben pasarse como resultado al siguiente; con casos simples, estos pueden ser reemplazados por una sola lista de comprensión. Además, podemos decir fácilmente de la comprensión de la lista cuál es el resultado de nuestro proceso, dónde hay más carga cognitiva al razonar sobre el proceso de Mapa y Filtro encadenado.

```
# Map & Filter
filtered = filter(lambda x: x % 2 == 0, range(10))
results = map(lambda x: 2*x, filtered)

# List comprehension
results = [2*x for x in range(10) if x % 2 == 0]
```

Refactorización - Referencia rápida

- **Mapa**

  ```
  map(F, S) == [F(x) for x in S]
  ```

- **Filtrar**

  ```
  filter(P, S) == [x for x in S if P(x)]
  ```

donde `F` y `P` son funciones que transforman respectivamente los valores de entrada y devuelven un `bool`

Comprensiones de lista anidadas

Las comprensiones de listas anidadas, a diferencia de las comprensiones de listas con bucles anidados, son comprensiones de listas dentro de una comprensión de listas. La expresión inicial puede ser cualquier expresión arbitraria, incluida otra lista de comprensión.

```
#List Comprehension with nested loop
[x + y for x in [1, 2, 3] for y in [3, 4, 5]]
#Out: [4, 5, 6, 5, 6, 7, 6, 7, 8]

#Nested List Comprehension
[[x + y for x in [1, 2, 3]] for y in [3, 4, 5]]
#Out: [[4, 5, 6], [5, 6, 7], [6, 7, 8]]
```

El ejemplo anidado es equivalente a

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

Un ejemplo donde se puede usar una comprensión anidada para transponer una matriz.

```
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]

[[row[i] for row in matrix] for i in range(len(matrix))]
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Al igual que anidado `for` bucles, no hay límite a cómo se pueden anidar las comprensiones profundas.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [[['1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

Iterar dos o más listas simultáneamente dentro de la comprensión de la lista

Para iterar más de dos listas simultáneamente dentro de la *comprensión de la lista* , se puede usar `zip()` como:

```
>>> list_1 = [1, 2, 3 , 4]
>>> list_2 = ['a', 'b', 'c', 'd']
>>> list_3 = ['6', '7', '8', '9']

# Two lists
>>> [(i, j) for i, j in zip(list_1, list_2)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# Three lists
```

```
>>> [(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]

# so on ...
```

Lea Lista de Comprensiones en línea: https://riptutorial.com/es/python/topic/5265/lista-de-comprensiones

Capítulo 113: Lista de desestructuración (también conocido como embalaje y desembalaje)

Examples

Tarea de destrucción

En las asignaciones, puede dividir un Iterable en valores usando la sintaxis de "desempaquetar":

La destrucción como valores.

```
a, b = (1, 2)
print(a)
# Prints: 1
print(b)
# Prints: 2
```

Si intenta desempaquetar más de la longitud del iterable, obtendrá un error:

```
a, b, c = [1]
# Raises: ValueError: not enough values to unpack (expected 3, got 1)
```

Python 3.x 3.0

La destrucción como lista

Puede desempaquetar una lista de longitud desconocida usando la siguiente sintaxis:

```
head, *tail = [1, 2, 3, 4, 5]
```

Aquí, extraemos el primer valor como un escalar, y los otros valores como una lista:

```
print(head)
# Prints: 1
print(tail)
# Prints: [2, 3, 4, 5]
```

Lo que equivale a:

```
l = [1, 2, 3, 4, 5]
head = l[0]
tail = l[1:]
```

También funciona con múltiples elementos o elementos del final de la lista:

```
a, b, *other, z = [1, 2, 3, 4, 5]
print(a, b, z, other)
# Prints: 1 2 5 [3, 4]
```

Ignorar valores en las tareas de desestructuración.

Si solo está interesado en un valor dado, puede usar _ para indicar que no está interesado. Nota: esto aún establecerá _ , solo la mayoría de la gente no lo usa como una variable.

```
a, _ = [1, 2]
print(a)
# Prints: 1
a, _, c = (1, 2, 3)
print(a)
# Prints: 1
print(c)
# Prints: 3
```

Python 3.x 3.0

Ignorar listas en tareas de desestructuración.

Finalmente, puede ignorar muchos valores usando la sintaxis `*_` en la asignación:

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Prints: 1
```

lo que no es realmente interesante, ya que podría usar la indexación en la lista. Donde se pone agradable es mantener el primer y último valor en una tarea:

```
 a, *_, b = [1, 2, 3, 4, 5]
 print(a, b)
 # Prints: 1 5
```

o extraer varios valores a la vez:

```
 a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
 print(a, b, c)
 # Prints: 1 3 5
```

Argumentos de la función de embalaje

En funciones, puede definir una serie de argumentos obligatorios:

```
def fun1(arg1, arg2, arg3):
    return (arg1,arg2,arg3)
```

lo que hará que la función se pueda llamar solo cuando se den los tres argumentos:

```
fun1(1, 2, 3)
```

y puede definir los argumentos como opcionales, utilizando valores predeterminados:

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1,arg2,arg3)
```

por lo que puede llamar a la función de muchas maneras diferentes, como:

```
fun2(1)              → (1,b,c)
fun2(1, 2)           → (1,2,c)
fun2(arg2=2, arg3=3) → (a,2,3)
...
```

Pero también puede usar la sintaxis de desestructuración para *empacar* argumentos, de modo que puede asignar variables usando una `list` o un `dict` .

Empaquetando una lista de argumentos

Considera que tienes una lista de valores.

```
l = [1,2,3]
```

Puede llamar a la función con la lista de valores como un argumento usando la sintaxis `*` :

```
fun1(*l)
# Returns: (1,2,3)
fun1(*['w', 't', 'f'])
# Returns: ('w','t','f')
```

Pero si no proporciona una lista cuya longitud coincida con el número de argumentos:

```
fun1(*['oops'])
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3'
```

Packing argumentos de palabras clave

Ahora, también puede empaquetar argumentos utilizando un diccionario. Puede usar el operador `**` para decirle a Python que descomprima el `dict` como valores de parámetros:

```
d = {
  'arg1': 1,
  'arg2': 2,
  'arg3': 3
}
fun1(**d)
# Returns: (1, 2, 3)
```

cuando la función solo tiene argumentos posicionales (los que no tienen valores predeterminados), necesita que el diccionario contenga todos los parámetros esperados y no tenga un parámetro adicional, o obtendrá un error:

```
fun1(**{'arg1':1, 'arg2':2})
# Raises: TypeError: fun1() missing 1 required positional argument: 'arg3'
fun1(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun1() got an unexpected keyword argument 'arg4'
```

Para las funciones que tienen argumentos opcionales, puede empaquetar los argumentos como un diccionario de la misma manera:

```
fun2(**d)
# Returns: (1, 2, 3)
```

Pero allí puede omitir valores, ya que serán reemplazados por los valores predeterminados:

```
fun2(**{'arg2': 2})
# Returns: ('a', 2, 'c')
```

Y al igual que antes, no puede dar valores adicionales que no sean parámetros existentes:

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun2() got an unexpected keyword argument 'arg4'
```

En el uso del mundo real, las funciones pueden tener argumentos tanto posicionales como opcionales, y funciona de la misma manera:

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

Puedes llamar a la función con solo un iterable:

```
fun3(*[1])
# Returns: (1, 'b', 'c')
fun3(*[1,2,3])
# Returns: (1, 2, 3)
```

o con solo un diccionario:

```
fun3(**{'arg1':1})
# Returns: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Returns: (1, 2, 3)
```

o puedes usar ambos en la misma llamada:

```
fun3(*[1,2], **{'arg3':3})
# Returns: (1,2,3)
```

Tenga en cuenta que no puede proporcionar varios valores para el mismo argumento:

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# Raises: TypeError: fun3() got multiple values for argument 'arg2'
```

Desempaquetando argumentos de funciones

Cuando desee crear una función que pueda aceptar cualquier número de argumentos y no imponer la posición o el nombre del argumento en el momento de la "compilación", es posible y a continuación le indicamos cómo:

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

Los parámetros `*args` y `**kwargs` son parámetros especiales que se establecen en una `tuple` y un `dict` , respectivamente:

```
fun1(1,2,3)
# Prints: (1, 2, 3) {}
fun1(a=1, b=2, c=3)
# Prints: () {'a': 1, 'b': 2, 'c': 3}
fun1('x', 'y', 'z', a=1, b=2, c=3)
# Prints: ('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
```

Si observa suficiente código de Python, descubrirá rápidamente que se está utilizando ampliamente al pasar argumentos a otra función. Por ejemplo, si desea extender la clase de cadena:

```
class MyString(str):
    def __init__(self, *args, **kwarg):
        print('Constructing MyString')
        super(MyString, self).__init__(*args, **kwarg)
```

Lea Lista de desestructuración (también conocido como embalaje y desembalaje) en línea: https://riptutorial.com/es/python/topic/4282/lista-de-desestructuracion--tambien-conocido-como-embalaje-y-desembalaje-

Capítulo 114: Listar comprensiones

Introducción

Las comprensiones de listas en Python son construcciones sintácticas concisas. Se pueden utilizar para generar listas de otras listas aplicando funciones a cada elemento de la lista. La siguiente sección explica y demuestra el uso de estas expresiones.

Sintaxis

- [x + 1 para x en (1, 2, 3)] # comprensión de lista, da [2, 3, 4]
- (x + 1 para x en (1, 2, 3)) # expresión del generador, producirá 2, luego 3, luego 4
- [x para x en (1, 2, 3) si x% 2 == 0] # enumerar comprensión con filtro, da [2]
- [x + 1 si x% 2 == 0 sino x para x en (1, 2, 3)] # lista comprensión con ternario
- [x + 1 si x% 2 == 0 sino x para x en el rango (-3,4) si x> 0] # lista comprensión con ternario y filtrado
- {x para x en (1, 2, 2, 3)} # establecer comprensión, da {1, 2, 3}
- {k: v para k, v en [('a', 1), ('b', 2)]} # dict comprensión, da {'a': 1, 'b': 2} (python 2.7+ y 3.0+ solamente)
- [x + y para x en [1, 2] para y en [10, 20]] # Bucles anidados, da [11, 21, 12, 22]
- [x + y para x en [1, 2, 3] si x> 2 para y en [3, 4, 5]] # Condición verificada al principio para bucle
- [x + y para x en [1, 2, 3] para y en [3, 4, 5] si x> 2] # Condición verificada en 2da para bucle
- [x para x en xrange (10) si x% 2 == 0] # Condición verificada si los números en bucle son números impares

Observaciones

Las comprensiones son construcciones sintácticas que definen estructuras de datos o expresiones exclusivas de un idioma en particular. El uso adecuado de las comprensiones las reinterpreta en expresiones fáciles de entender. Como expresiones, se pueden utilizar:

- en el lado derecho de las tareas
- como argumentos para llamadas de funcion
- en el cuerpo de una función lambda
- como declaraciones independientes. (Por ejemplo: `[print(x) for x in range(10)]`)

Examples

Lista de Comprensiones

Una lista de comprensión crea una nueva `list` al aplicar una expresión a cada elemento de un iterable . La forma más básica es:

```
[ <expression> for <element> in <iterable> ]
```

También hay una condición opcional "si":

```
[ <expression> for <element> in <iterable> if <condition> ]
```

Cada `<element>` en el `<iterable>` se conecta a la `<expression>` si el `<condition>` (opcional) se evalúa como verdadero . Todos los resultados se devuelven a la vez en la nueva lista. Las expresiones de los generadores se evalúan perezosamente, pero las comprensiones de la lista evalúan todo el iterador inmediatamente, consumiendo memoria proporcional a la longitud del iterador.

Para crear una `list` de enteros cuadrados:

```
squares = [x * x for x in (1, 2, 3, 4)]
# squares: [1, 4, 9, 16]
```

La expresión `for` establece `x` a cada valor por turno de `(1, 2, 3, 4)` . El resultado de la expresión `x * x` se anexa a una `list` interna. La `list` interna se asigna a los `squares` variables cuando se completa.

Además de un aumento de velocidad (como se explica aquí), una comprensión de la lista es aproximadamente equivalente al siguiente bucle for:

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
# squares: [1, 4, 9, 16]
```

La expresión aplicada a cada elemento puede ser tan compleja como sea necesario:

```
# Get a list of uppercase characters from a string
[s.upper() for s in "Hello World"]
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']

# Strip off any commas from the end of strings in a list
[w.strip(',') for w in ['these,', 'words,,', 'mostly', 'have,commas,']]
# ['these', 'words', 'mostly', 'have,commas']

# Organize letters in words more reasonably - in an alphabetical order
sentence = "Beautiful is better than ugly"
["".join(sorted(word, key = lambda x: x.lower())) for word in sentence.split()]
# ['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

---

más

`else` se puede utilizar en las construcciones de comprensión de listas, pero tenga cuidado con la sintaxis. Las cláusulas if / else deben usarse antes `for` bucle, no después:

```
# create a list of characters in apple, replacing non vowels with '*'
# Ex - 'apple' --> ['a', '*', '*', '*' ,'e']

[x for x in 'apple' if x in 'aeiou' else '*']
#SyntaxError: invalid syntax

# When using if/else together use them before the loop
[x if x in 'aeiou' else '*' for x in 'apple']
#['a', '*', '*', '*', 'e']
```

Tenga en cuenta que esto utiliza una construcción de lenguaje diferente, una expresión condicional , que en sí misma no es parte de la sintaxis de comprensión . Considerando que el `if` after the `for…in` *es* parte de la lista de comprensión y se utiliza para *filtrar* elementos de la fuente iterable.

Doble iteración

El orden de doble iteración `[... for x in ... for y in ...]` es natural o contraintuitivo. La regla de oro es seguir un equivalente `for` bucle:

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```

Esto se convierte en:

```
[str(x)
    for i in range(3)
        for x in foo(i)
]
```

Esto se puede comprimir en una línea como `[str(x) for i in range(3) for x in foo(i)]`

Mutación in situ y otros efectos secundarios

Antes de usar la comprensión de lista, comprenda la diferencia entre las funciones solicitadas por sus efectos secundarios (funciones *mutantes* o *en el lugar*) que generalmente devuelven `None` y las funciones que devuelven un valor interesante.

Muchas funciones (especialmente funciones *puras*) simplemente toman un objeto y devuelven algún objeto. Una función *en el lugar* modifica el objeto existente, que se denomina *efecto secundario* . Otros ejemplos incluyen operaciones de entrada y salida, como la impresión.

`list.sort()` ordena una lista *en el lugar* (lo que significa que modifica la lista original) y devuelve el

valor `None` . Por lo tanto, no funcionará como se espera en una lista de comprensión:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

En su lugar, `sorted()` devuelve una `list` ordenada en lugar de ordenar in situ:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

Es posible el uso de comprensiones para efectos secundarios, como I / O o funciones in situ. Sin embargo, un bucle for suele ser más legible. Mientras esto funciona en Python 3:

```
[print(x) for x in (1, 2, 3)]
```

En su lugar, utilice:

```
for x in (1, 2, 3):
    print(x)
```

En algunas situaciones, las funciones de efectos secundarios *son* adecuadas para la comprensión de listas. `random.randrange()` tiene el efecto secundario de cambiar el estado del generador de números aleatorios, pero también devuelve un valor interesante. Además, se puede llamar a `next()` en un iterador.

El siguiente generador de valores aleatorios no es puro, pero tiene sentido ya que el generador aleatorio se restablece cada vez que se evalúa la expresión:

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

Los espacios en blanco en la lista de comprensión

Las comprensiones de listas más complicadas pueden alcanzar una longitud no deseada o volverse menos legibles. Aunque es menos común en los ejemplos, es posible dividir una lista de comprensión en varias líneas, como por ejemplo:

```
[
    x for x
    in 'foo'
    if x not in 'bar'
]
```

Diccionario de Comprensiones

Una comprensión de diccionario es similar a una comprensión de lista, excepto que produce un objeto de diccionario en lugar de una lista.

Un ejemplo básico:

Python 2.x 2.7

```
{x: x * x for x in (1, 2, 3, 4)}
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

que es solo otra forma de escribir:

```
dict((x, x * x) for x in (1, 2, 3, 4))
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

Al igual que con una lista de comprensión, podemos usar una declaración condicional dentro de la comprensión de dict para producir solo los elementos de dict que cumplan con algún criterio.

Python 2.x 2.7

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# Out: {'Exchange': 8, 'Overflow': 8}
```

O, reescrito usando una expresión generadora.

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# Out: {'Exchange': 8, 'Overflow': 8}
```

Comenzando con un diccionario y utilizando la comprensión del diccionario como un filtro de par clave-valor

Python 2.x 2.7

```
initial_dict = {'x': 1, 'y': 2}
{key: value for key, value in initial_dict.items() if key == 'x'}
# Out: {'x': 1}
```

Tecla de conmutación y valor del diccionario (diccionario invertido)

Si tiene un dict que contiene valores *hashables* simples (los valores duplicados pueden tener resultados inesperados):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

y quería intercambiar las claves y los valores, puede adoptar varios enfoques dependiendo de su

estilo de codificación:

- `swapped = {v: k for k, v in my_dict.items()}`
- `swapped = dict((v, k) for k, v in my_dict.iteritems())`
- `swapped = dict(zip(my_dict.values(), my_dict))`
- `swapped = dict(zip(my_dict.values(), my_dict.keys()))`
- `swapped = dict(map(reversed, my_dict.items()))`

```
print(swapped)
# Out: {a: 1, b: 2, c: 3}
```

Python 2.x 2.3

Si su diccionario es grande, considere *la importación* *itertools* y utilizar `izip` o `imap` .

Fusionando diccionarios

Combine diccionarios y, opcionalmente, anule valores antiguos con una comprensión de diccionario anidado.

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Sin embargo, el desempaque del diccionario (PEP 448) puede ser el preferido.

Python 3.x 3.5

```
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Nota : las comprensiones de diccionarios se agregaron en Python 3.0 y se respaldaron a 2.7+, a diferencia de las comprensiones de listas, que se agregaron en 2.0. Las versiones <2.7 pueden usar expresiones generadoras y el `dict()` incorporado para simular el comportamiento de las comprensiones de diccionario.

Expresiones del generador

Las expresiones generadoras son muy similares a las listas de comprensión. La principal diferencia es que no crea un conjunto completo de resultados a la vez; crea un objeto generador que luego puede ser iterado.

Por ejemplo, vea la diferencia en el siguiente código:

```
# list comprehension
[x**2 for x in range(10)]
# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python 2.x 2.4

```
# generator comprehension
(x**2 for x in xrange(10))
# Output: <generator object <genexpr> at 0x11b4b7c80>
```

Estos son dos objetos muy diferentes:

- la lista de comprensión devuelve un objeto de `list` mientras que la comprensión del generador devuelve un `generator` .
- `generator` objetos `generator` no se pueden indexar y hace uso de la `next` función para ordenar los artículos.

Nota : Utilizamos `xrange` ya que también crea un objeto generador. Si usaríamos el rango, se crearía una lista. Además, `xrange` solo existe en la versión posterior de python 2. En python 3, `range` solo devuelve un generador. Para obtener más información, consulte el ejemplo de *Diferencias entre las funciones de rango y rango* .

Python 2.x 2.4

```
g = (x**2 for x in xrange(10))
print(g[0])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object has no attribute '__getitem__'
```

```
g.next()  # 0
g.next()  # 1
g.next()  # 4
...
g.next()  # 81

g.next()  # Throws StopIteration Exception
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Python 3.x 3.0

> NOTA: La función `g.next()` debe sustituirse por `next(g)` y `xrange` con `range` ya que `Iterator.next()` y `xrange()` no existen en Python 3.

Aunque ambos pueden ser iterados de manera similar:

```
for i in [x**2 for x in range(10)]:
    print(i)

"""
```

```
Out:
0
1
4
...
81
"""
```

Python 2.x 2.4

```
for i in (x**2 for x in xrange(10)):
    print(i)

"""
Out:
0
1
4
.
.
.
81
"""
```

Casos de uso

Las expresiones del generador se evalúan perezosamente, lo que significa que generan y devuelven cada valor solo cuando el generador está iterado. Esto suele ser útil cuando se itera a través de grandes conjuntos de datos, evitando la necesidad de crear un duplicado del conjunto de datos en la memoria:

```
for square in (x**2 for x in range(1000000)):
    #do something
```

Otro caso de uso común es evitar la iteración en todo un iterable si no es necesario hacerlo. En este ejemplo, un elemento se recupera de una API remota con cada iteración de `get_objects()` . Pueden existir miles de objetos, deben recuperarse uno por uno, y solo necesitamos saber si existe un objeto que coincida con un patrón. Al usar una expresión generadora, cuando encontramos un objeto que coincide con el patrón.

```
def get_objects():
    """Gets objects from an API one by one"""
    while True:
        yield get_next_item()

def object_matches_pattern(obj):
    # perform potentially complex calculation
    return matches_pattern

def right_item_exists():
    items = (object_matched_pattern(each) for each in get_objects())
    for item in items:
        if item.is_the_right_one:
```

```
            return True
    return False
```

Establecer Comprensiones

La comprensión del conjunto es similar a la lista y la comprensión del diccionario , pero produce un conjunto , que es una colección desordenada de elementos únicos.

Python 2.x 2.7

```
# A set containing every value in range(5):
{x for x in range(5)}
# Out: {0, 1, 2, 3, 4}

# A set of even numbers between 1 and 10:
{x for x in range(1, 11) if x % 2 == 0}
# Out: {2, 4, 6, 8, 10}

# Unique alphabetic characters in a string of text:
text = "When in the Course of human events it becomes necessary for one people..."
{ch.lower() for ch in text if ch.isalpha()}
# Out: set(['a', 'c', 'b', 'e', 'f', 'i', 'h', 'm', 'l', 'o',
#           'n', 'p', 's', 'r', 'u', 't', 'w', 'v', 'y'])
```

Demo en vivo

Tenga en cuenta que los conjuntos están desordenados. Esto significa que el orden de los resultados en el conjunto puede diferir del presentado en los ejemplos anteriores.

Nota : la comprensión de configuración está disponible desde python 2.7+, a diferencia de las comprensiones de lista, que se agregaron en 2.0. En Python 2.2 a Python 2.6, la función `set()` se puede usar con una expresión generadora para producir el mismo resultado:

Python 2.x 2.2

```
set(x for x in range(5))
# Out: {0, 1, 2, 3, 4}
```

Evite operaciones repetitivas y costosas usando cláusula condicional

Considere la siguiente lista de comprensión:

```
>>> def f(x):
...     import time
...     time.sleep(.1)       # Simulate expensive function
...     return x**2

>>> [f(x) for x in range(1000) if f(x) > 10]
[16, 25, 36, ...]
```

Esto da como resultado dos llamadas a `f(x)` para 1,000 valores de `x` : una llamada para generar

el valor y la otra para verificar la condición `if` . Si `f(x)` es una operación particularmente costosa, esto puede tener implicaciones significativas en el rendimiento. Peor aún, si llamar a `f()` tiene efectos secundarios, puede tener resultados sorprendentes.

En su lugar, debe evaluar la operación costosa solo una vez para cada valor de `x` generando un iterable intermedio (expresión del generador) de la siguiente manera:

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

O, usando el mapa incorporado equivalente:

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

Otra forma que podría resultar en un código más legible es colocar el resultado parcial (`v` en el ejemplo anterior) en un iterable (como una lista o una tupla) y luego iterar sobre él. Como `v` será el único elemento en el iterable, el resultado es que ahora tenemos una referencia a la salida de nuestra función lenta calculada solo una vez:

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
[16, 25, 36, ...]
```

Sin embargo, en la práctica, la lógica del código puede ser más complicada y es importante mantenerlo legible. En general, se recomienda una función de generador por separado en un complejo de una sola línea:

```
>>> def process_prime_numbers(iterable):
...     for x in iterable:
...         if is_prime(x):
...             yield f(x)
...
>>> [x for x in process_prime_numbers(range(1000)) if x > 10]
[11, 13, 17, 19, ...]
```

Otra manera de evitar el cálculo de `f(x)` varias veces es utilizar el `@functools.lru_cache()` (Python 3.2+) decorador en `f(x)` . De esta manera, dado que la salida de `f` para la entrada `x` ya se ha calculado una vez, la invocación de la segunda función de la comprensión de la lista original será tan rápida como la búsqueda de un diccionario. Este enfoque utiliza la memoria para mejorar la eficiencia, que es comparable al uso de expresiones generadoras.

Di que tienes que aplanar una lista

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Algunos de los métodos podrían ser:

```
reduce(lambda x, y: x+y, l)

sum(l, [])

list(itertools.chain(*l))
```

Sin embargo, la comprensión de la lista proporcionaría la mejor complejidad de tiempo.

```
[item for sublist in l for item in sublist]
```

Los accesos directos basados en + (incluido el uso implícito en la suma) son, por necesidad, O (L ^ 2) cuando hay L sublistas: a medida que la lista de resultados intermedios se hace más larga, en cada paso se obtiene un nuevo objeto de lista de resultados intermedios. asignados, y todos los elementos en el resultado intermedio anterior deben copiarse (así como algunos nuevos agregados al final). Entonces (por simplicidad y sin pérdida de generalidad real) digamos que tiene L sublistas de I elementos cada uno: los primeros elementos I se copian una y otra vez L-1 veces, el segundo I elementos L-2 veces, y así sucesivamente; el número total de copias es I veces la suma de x para x de 1 a L excluidas, es decir, I * (L ** 2) / 2.

La lista de comprensión solo genera una lista, una vez, y copia cada elemento (de su lugar de residencia original a la lista de resultados) también una sola vez.

Comprensiones que involucran tuplas

La cláusula `for` de una lista de comprensión puede especificar más de una variable:

```
[x + y for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [3, 7, 11]

[x + y for x, y in zip([1, 3, 5], [2, 4, 6])]
# Out: [3, 7, 11]
```

Esto es como regular `for` bucles:

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

Sin embargo, tenga en cuenta que si la expresión que comienza la comprensión es una tupla, debe estar entre paréntesis:

```
[x, y for x, y in [(1, 2), (3, 4), (5, 6)]]
# SyntaxError: invalid syntax

[(x, y) for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [(1, 2), (3, 4), (5, 6)]
```

Contando Ocurrencias Usando Comprensión

Cuando queremos contar el número de elementos en un iterable, que cumplan con alguna condición, podemos usar la comprensión para producir una sintaxis idiomática:

```
# Count the numbers in `range(1000)` that are even and contain the digit `9`:
print (sum(
    1 for x in range(1000)
    if x % 2 == 0 and
    '9' in str(x)
))
# Out: 95
```

El concepto básico se puede resumir como:

1. Iterar sobre los elementos en `range(1000)` .
2. Concatenar todo lo necesario `if` condiciones.
3. Utilice 1 como *expresión* para devolver un 1 por cada elemento que cumpla con las condiciones.
4. Resuma todos los `1` s para determinar la cantidad de elementos que cumplen con las condiciones.

Nota : Aquí no estamos recolectando los `1` s en una lista (note la ausencia de corchetes), pero pasamos los directamente a la función de `sum` que los está sumando. Esto se denomina *expresión generadora* , que es similar a una comprensión.

Cambio de tipos en una lista

Los datos cuantitativos a menudo se leen como cadenas que deben convertirse en tipos numéricos antes de procesarlos. Los tipos de todos los elementos de la lista se pueden convertir con una Comprensión de lista o la función `map()` .

```
# Convert a list of strings to integers.
items = ["1","2","3","4"]
[int(item) for item in items]
# Out: [1, 2, 3, 4]

# Convert a list of strings to float.
items = ["1","2","3","4"]
map(float, items)
# Out:[1.0, 2.0, 3.0, 4.0]
```

Lea Listar comprensiones en línea: https://riptutorial.com/es/python/topic/196/listar-comprensiones

Capítulo 115: Listas enlazadas

Introducción

Una lista enlazada es una colección de nodos, cada uno compuesto por una referencia y un valor. Los nodos se unen en una secuencia utilizando sus referencias. Las listas vinculadas se pueden utilizar para implementar estructuras de datos más complejas como listas, pilas, colas y matrices asociativas.

Examples

Ejemplo de lista enlazada única

Este ejemplo implementa una lista enlazada con muchos de los mismos métodos que el objeto de lista integrado.

```
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, val):
        self.data = val

    def setNext(self, val):
        self.next = val

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        """Check if the list is empty"""
        return self.head is None

    def add(self, item):
        """Add the item to the list"""
        new_node = Node(item)
        new_node.setNext(self.head)
        self.head = new_node

    def size(self):
        """Return the length/size of the list"""
        count = 0
        current = self.head
        while current is not None:
            count += 1
```

```
            current = current.getNext()
        return count

    def search(self, item):
        """Search for item in list. If found, return True. If not found, return False"""
        current = self.head
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
        return found

    def remove(self, item):
        """Remove item from list. If item is not found in list, raise ValueError"""
        current = self.head
        previous = None
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if found:
            if previous is None:
                self.head = current.getNext()
            else:
                previous.setNext(current.getNext())
        else:
            raise ValueError
            print 'Value not found.'

    def insert(self, position, item):
        """
        Insert item at position specified. If position specified is
        out of bounds, raise IndexError
        """
        if position > self.size() - 1:
            raise IndexError
            print "Index out of bounds."
        current = self.head
        previous = None
        pos = 0
        if position is 0:
            self.add(item)
        else:
            new_node = Node(item)
            while pos < position:
                pos += 1
                previous = current
                current = current.getNext()
            previous.setNext(new_node)
            new_node.setNext(current)

    def index(self, item):
        """
        Return the index where item is found.
        If item is not found, return None.
        """
```

```
        current = self.head
        pos = 0
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
                pos += 1
        if found:
            pass
        else:
            pos = None
        return pos

    def pop(self, position = None):
        """
        If no argument is provided, return and remove the item at the head.
        If position is provided, return and remove the item at that position.
        If index is out of bounds, raise IndexError
        """
        if position > self.size():
            print 'Index out of bounds'
            raise IndexError

        current = self.head
        if position is None:
            ret = current.getData()
            self.head = current.getNext()
        else:
            pos = 0
            previous = None
            while pos < position:
                previous = current
                current = current.getNext()
                pos += 1
                ret = current.getData()
            previous.setNext(current.getNext())
        print ret
        return ret

    def append(self, item):
        """Append item to the end of the list"""
        current = self.head
        previous = None
        pos = 0
        length = self.size()
        while pos < length:
            previous = current
            current = current.getNext()
            pos += 1
        new_node = Node(item)
        if previous is None:
            new_node.setNext(current)
            self.head = new_node
        else:
            previous.setNext(new_node)

    def printList(self):
        """Print the list"""
        current = self.head
```

```
        while current is not None:
            print current.getData()
            current = current.getNext()
```

El uso funciona muy parecido al de la lista incorporada.

```
ll = LinkedList()
ll.add('l')
ll.add('H')
ll.insert(1,'e')
ll.append('l')
ll.append('o')
ll.printList()

H
e
l
l
o
```

Lea Listas enlazadas en línea: https://riptutorial.com/es/python/topic/9299/listas-enlazadas

Capítulo 116: Llama a Python desde C

Introducción

La documentación proporciona una implementación de muestra de la comunicación entre procesos entre C # y los scripts de Python.

Observaciones

Tenga en cuenta que en el ejemplo anterior, los datos se serializan utilizando la biblioteca **MongoDB.Bson** que se puede instalar a través del administrador NuGet.

De lo contrario, puede utilizar cualquier biblioteca de serialización JSON de su elección.

A continuación se presentan los pasos de implementación de la comunicación entre procesos:

- Los argumentos de entrada se serializan en una cadena JSON y se guardan en un archivo de texto temporal:

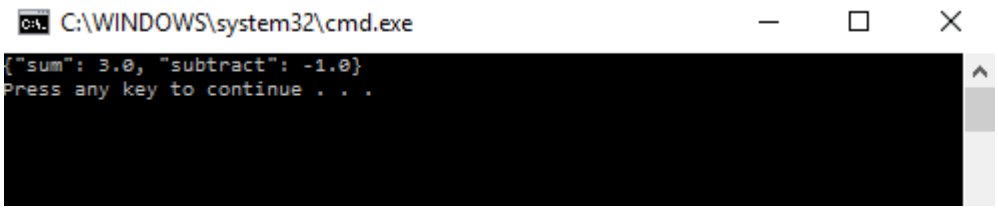
```
 BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");
 string argsFile = string.Format("{0}\\{1}.txt", Path.GetDirectoryName(pyScriptPath),
Guid.NewGuid());
```

- Python interpreter python.exe ejecuta la secuencia de comandos de python que lee la cadena JSON desde un archivo de texto temporal y los argumentos de entrada de retroceso:

```
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
   input_args = json.loads( data_file.read() )

x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]
```

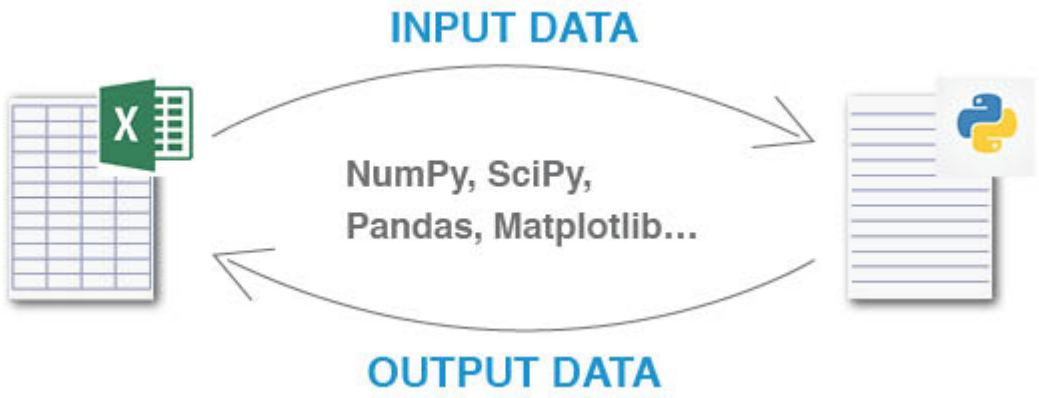
- El script de Python se ejecuta y el diccionario de salida se serializa en una cadena JSON y se imprime en la ventana de comandos:

```
print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```



- Lea la cadena JSON de salida de la aplicación C #:

```
using (StreamReader myStreamReader = process.StandardOutput)
{
   outputString = myStreamReader.ReadLine();
   process.WaitForExit();
}
```



Estoy utilizando la comunicación entre procesos entre C # y los scripts de Python en uno de mis proyectos que permite llamar a los scripts de Python directamente desde hojas de cálculo de Excel.

El proyecto utiliza el complemento ExcelDNA para C # - enlace Excel.

El código fuente se almacena en el repositorio de GitHub.

A continuación se encuentran enlaces a páginas wiki que brindan una descripción general del proyecto y ayudan a comenzar en 4 sencillos pasos .

- Empezando
- Descripción general de la implementación
- Ejemplos
- Asistente de objetos
- Funciones

Espero que encuentre útil el ejemplo y el proyecto.

Examples

Script de Python para ser llamado por la aplicación C

```
import sys
import json

# load input arguments from the text file
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
    input_args = json.loads( data_file.read() )

# cast strings to floats
```

```
x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]

print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```

Código C # llamando al script Python

```
using MongoDB.Bson;
using System;
using System.Diagnostics;
using System.IO;

namespace python_csharp
{
    class Program
    {
        static void Main(string[] args)
        {
            // full path to .py file
            string pyScriptPath = "...../sum.py";
            // convert input arguments to JSON string
            BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");

            bool saveInputFile = false;

            string argsFile = string.Format("{0}\\{1}.txt",
Path.GetDirectoryName(pyScriptPath), Guid.NewGuid());

            string outputString = null;
            // create new process start info
            ProcessStartInfo prcStartInfo = new ProcessStartInfo
            {
                // full path of the Python interpreter 'python.exe'
                FileName = "python.exe", // string.Format(@"""{0}""", "python.exe"),
                UseShellExecute = false,
                RedirectStandardOutput = true,
                CreateNoWindow = false
            };

            try
            {
                // write input arguments to .txt file
                using (StreamWriter sw = new StreamWriter(argsFile))
                {
                    sw.WriteLine(argsBson);
                    prcStartInfo.Arguments = string.Format("{0} {1}",
string.Format(@"""{0}""", pyScriptPath), string.Format(@"""{0}""", argsFile));
                }
                // start process
                using (Process process = Process.Start(prcStartInfo))
                {
                    // read standard output JSON string
                    using (StreamReader myStreamReader = process.StandardOutput)
                    {
                        outputString = myStreamReader.ReadLine();
                        process.WaitForExit();
                    }
                }
            }
            finally
```

```
            {
                // delete/save temporary .txt file
                if (!saveInputFile)
                {
                    File.Delete(argsFile);
                }
            }
            Console.WriteLine(outputString);
        }
    }
}
```

Lea Llama a Python desde C # en línea: https://riptutorial.com/es/python/topic/10759/llama-a-python-desde-c-sharp

Capítulo 117: Maldiciones básicas con pitón

Observaciones

Curses es un módulo básico de manejo de terminal (o visualización de caracteres) de Python. Esto se puede utilizar para crear interfaces de usuario basadas en terminal o TUI.

Este es un puerto python de una biblioteca C más popular 'ncurses'

Examples

Ejemplo básico de invocación

```
import curses
import traceback

try:
    # -- Initialize --
    stdscr = curses.initscr()    # initialize curses screen
    curses.noecho()              # turn off auto echoing of keypress on to screen
    curses.cbreak()              # enter break mode where pressing Enter key
                                 #  after keystroke is not required for it to register
    stdscr.keypad(1)             # enable special Key values such as curses.KEY_LEFT etc

    # -- Perform an action with Screen --
    stdscr.border(0)
    stdscr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    stdscr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = stdscr.getch()
        if ch == ord('q'):
            break

    # -- End of user code --

except:
    traceback.print_exc()     # print trace back log of the error

finally:
    # --- Cleanup on exit ---
    stdscr.keypad(0)
    curses.echo()
    curses.nocbreak()
    curses.endwin()
```

La función de ayuda wrapper ().

Si bien la invocación básica anterior es bastante fácil, el paquete curses proporciona la `wrapper(func, ...)` ayuda de `wrapper(func, ...)` . El siguiente ejemplo contiene el equivalente de arriba:

```
main(scr, *args):
    # -- Perform an action with Screen --
    scr.border(0)
    scr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    scr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = scr.getch()
        if ch == ord('q'):

curses.wrapper(main)
```

Aquí, la envoltura inicializará las cursas, creará `stdscr` , un objeto WindowObject y pasará ambos stdscr, y cualquier argumento adicional a `func` . Cuando la `func` vuelve, la `wrapper` restaurará el terminal antes de que el programa salga.

Lea Maldiciones básicas con pitón en línea:
https://riptutorial.com/es/python/topic/5851/maldiciones-basicas-con-piton

Capítulo 118: Manipulando XML

Observaciones

No todos los elementos de la entrada XML terminarán como elementos del árbol analizado. Actualmente, este módulo omite los comentarios XML, las instrucciones de procesamiento y las declaraciones de tipo de documento en la entrada. Sin embargo, los árboles construidos utilizando la API de este módulo en lugar de analizar a partir de texto XML pueden tener comentarios e instrucciones de procesamiento; Se incluirán al generar la salida XML.

Examples

Abriendo y leyendo usando un ElementTree

Importe el objeto ElementTree, abra el archivo .xml relevante y obtenga la etiqueta raíz:

```
import xml.etree.ElementTree as ET
tree = ET.parse("yourXMLfile.xml")
root = tree.getroot()
```

Hay algunas maneras de buscar a través del árbol. Primero es por iteración:

```
for child in root:
    print(child.tag, child.attrib)
```

De lo contrario, puede hacer referencia a ubicaciones específicas como una lista:

```
print(root[0][1].text)
```

Para buscar etiquetas específicas por nombre, use `.find` o `.findall` :

```
print(root.findall("myTag"))
print(root[0].find("myOtherTag"))
```

Modificar un archivo XML

Importar módulo de árbol de elementos y abrir archivo xml, obtener un elemento xml

```
import xml.etree.ElementTree as ET
tree = ET.parse('sample.xml')
root=tree.getroot()
element = root[0] #get first child of root element
```

El objeto elemento se puede manipular cambiando sus campos, agregando y modificando atributos, agregando y eliminando elementos secundarios

```
element.set('attribute_name', 'attribute_value') #set the attribute to xml element
element.text="string_text"
```

Si desea eliminar un elemento use el método Element.remove ()

```
root.remove(element)
```

Método ElementTree.write () utilizado para generar un objeto xml en archivos xml.

```
tree.write('output.xml')
```

Crear y construir documentos XML

Módulo de árbol de elementos de importación

```
import xml.etree.ElementTree as ET
```

La función Element () se utiliza para crear elementos XML

```
p=ET.Element('parent')
```

Función de elemento secundario () utilizada para crear subelementos a un elemento dado

```
c = ET.SubElement(p, 'child1')
```

La función dump () se utiliza para volcar elementos xml.

```
ET.dump(p)
# Output will be like this
#<parent><child1 /></parent>
```

Si desea guardar en un archivo, cree un árbol xml con la función ElementTree () y guarde en un archivo con el método write ()

```
tree = ET.ElementTree(p)
tree.write("output.xml")
```

La función Comentario () se utiliza para insertar comentarios en un archivo xml.

```
comment = ET.Comment('user comment')
p.append(comment) #this comment will be appended to parent element
```

Abrir y leer archivos XML grandes utilizando iterparse (análisis incremental)

A veces no queremos cargar el archivo XML completo para obtener la información que necesitamos. En estos casos, es útil poder cargar incrementalmente las secciones relevantes y luego eliminarlas cuando hayamos terminado. Con la función iterparse puede editar el árbol de

elementos que se almacena mientras se analiza el XML.

Importe el objeto ElementTree:

```
import xml.etree.ElementTree as ET
```

Abra el archivo .xml e itere sobre todos los elementos:

```
for event, elem in ET.iterparse("yourXMLfile.xml"):
    ... do something ...
```

Alternativamente, solo podemos buscar eventos específicos, como etiquetas de inicio / finalización o espacios de nombres. Si esta opción se omite (como anteriormente), solo se devuelven eventos "finales":

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
    ... do something ...
```

Aquí está el ejemplo completo que muestra cómo borrar elementos del árbol en memoria cuando terminemos con ellos:

```
for event, elem in ET.iterparse("yourXMLfile.xml", events=("start","end")):
    if elem.tag == "record_tag" and event == "end":
        print elem.text
        elem.clear()
    ... do something else ...
```

Buscando el XML con XPath

A partir de la versión 2.7, `ElementTree` tiene un mejor soporte para consultas XPath. XPath es una sintaxis que le permite navegar a través de un xml como SQL se utiliza para buscar a través de una base de datos. Tanto las funciones `find` y `findall` son compatibles con XPath. El siguiente XML se utilizará para este ejemplo.

```
 <Catalog>
    <Books>
        <Book id="1" price="7.95">
            <Title>Do Androids Dream of Electric Sheep?</Title>
            <Author>Philip K. Dick</Author>
        </Book>
        <Book id="5" price="5.95">
            <Title>The Colour of Magic</Title>
            <Author>Terry Pratchett</Author>
        </Book>
        <Book id="7" price="6.95">
            <Title>The Eye of The World</Title>
            <Author>Robert Jordan</Author>
        </Book>
    </Books>
</Catalog>
```

Buscando todos los libros:

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

Buscando el libro con el título = 'El color de la magia':

```
tree.find("Books/Book[Title='The Colour of Magic']")
# always use '' in the right side of the comparison
```

Buscando el libro con id = 5:

```
tree.find("Books/Book[@id='5']")
# searches with xml attributes must have '@' before the name
```

Busca el segundo libro:

```
tree.find("Books/Book[2]")
# indexes starts at 1, not 0
```

Buscar el último libro:

```
tree.find("Books/Book[last()]")
# 'last' is the only xpath function allowed in ElementTree
```

Búsqueda de todos los autores:

```
tree.findall(".//Author")
#searches with // must use a relative path
```

Lea Manipulando XML en línea: https://riptutorial.com/es/python/topic/479/manipulando-xml

Capítulo 119: Matemáticas complejas

Sintaxis

- cmath.rect (AbsoluteValue, Phase)

Examples

Aritmética compleja avanzada

El módulo `cmath` incluye funciones adicionales para usar números complejos.

```
import cmath
```

Este módulo puede calcular la fase de un número complejo, en radianes:

```
z = 2+3j # A complex number
cmath.phase(z) # 0.982793723247329
```

Permite la conversión entre las representaciones cartesianas (rectangulares) y polares de números complejos:

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)
cmath.rect(2, cmath.pi/2) # (0+2j)
```

El módulo contiene la versión compleja de

- Funciones exponenciales y logarítmicas (como es habitual, `log` es el logaritmo natural y `log10` el logaritmo decimal):

  ```
  cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)
  cmath.log(z) # (1.2824746787307684+0.982793723247329j)
  cmath.log10(-100) # (2+1.3643763538418412j)
  ```

- Raíces cuadradas:

  ```
  cmath.sqrt(z) # (1.6741492280355401+0.8959774761298381j)
  ```

- Funciones trigonométricas y sus inversos:

  ```
  cmath.sin(z)  # (9.15449914691143-4.168906959966565j)
  cmath.cos(z)  # (-4.189625690968807-9.109227893755337j)
  cmath.tan(z)  # (-0.003764025641504249+1.003238627353 61j)
  cmath.asin(z) # (0.5706527843210994+1.9833870299165355j)
  cmath.acos(z) # (1.0001435424737972-1.9833870299165355j)
  cmath.atan(z) # (1.4099210495965755+0.229072682968538 78j)
  cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
  ```

- Funciones hiperbólicas y sus inversos:

```
cmath.sinh(z)  # (-3.59056458998578+0.5309210862485197j)
cmath.cosh(z)  # (-3.7245455049153224+0.5118225699873846j)
cmath.tanh(z)  # (0.965385879022133-0.009884375038322495j)
cmath.asinh(z) # (0.5706527843210994+1.9833870299165355j)
cmath.acosh(z) # (1.9833870299165355+1.0001435424737972j)
cmath.atanh(z) # (0.14694666622552977+1.3389725222944935j)
cmath.cosh(z)**2 - cmath.sin(z)**2  # (1+0j)
cmath.cosh((0+1j)*z) - cmath.cos(z) # 0j
```

Aritmética compleja básica

Python ha incorporado soporte para aritmética compleja. La unidad imaginaria se denota por `j` :

```
z = 2+3j # A complex number
w = 1-7j # Another complex number
```

Los números complejos pueden sumarse, restarse, multiplicarse, dividirse y exponerse:

```
z + w # (3-4j)
z - w # (1+10j)
z * w # (23-11j)
z / w # (-0.38+0.34j)
z**3  # (-46+9j)
```

Python también puede extraer las partes reales e imaginarias de números complejos, y calcular su valor absoluto y su conjugado:

```
z.real # 2.0
z.imag # 3.0
abs(z) # 3.605551275463989
z.conjugate() # (2-3j)
```

Lea Matemáticas complejas en línea: https://riptutorial.com/es/python/topic/1142/matematicas-complejas

Capítulo 120: Matraz

Introducción

Flask es un marco micro web de Python que se utiliza para ejecutar los principales sitios web, incluidos Pintrest, Twilio y Linkedin. Este tema explica y demuestra la variedad de características que Flask ofrece para el desarrollo web tanto en la parte frontal como en la posterior.

Sintaxis

- @ app.route ("/ urlpath", métodos = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])
- @ app.route ("/ urlpath / <param>", methods = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])

Examples

Los basicos

El siguiente ejemplo es un ejemplo de un servidor básico:

```
# Imports the Flask class
from flask import Flask
# Creates an app and checks if its the main or imported
app = Flask(__name__)

# Specifies what URL triggers hello_world()
@app.route('/')
# The function run on the index route
def hello_world():
    # Returns the text to be displayed
    return "Hello World!"

# If this script isn't an import
if __name__ == "__main__":
    # Run the app until stopped
    app.run()
```

La ejecución de este script (con todas las dependencias correctas instaladas) debe iniciar un servidor local. El host es `127.0.0.1` comúnmente conocido como **localhost** . Este servidor se ejecuta por defecto en el puerto **5000** . Para acceder a su servidor web, abra un navegador web e ingrese la URL `localhost:5000` o `127.0.0.1:5000` (no hay diferencia). Actualmente, solo su computadora puede acceder al servidor web.

`app.run()` tiene tres parámetros, **host** , **puerto** y **depuración** . El host es `127.0.0.1` forma predeterminada, pero si lo establece en `0.0.0.0` podrá acceder a su servidor web desde cualquier dispositivo de su red usando su dirección IP privada en la URL. el puerto es por defecto 5000, pero si el parámetro se establece en el puerto `80` , los usuarios no necesitarán especificar un

número de puerto ya que los navegadores usan el puerto 80 de manera predeterminada. En cuanto a la opción de depuración, durante el proceso de desarrollo (nunca en producción), ayuda a establecer este parámetro en Verdadero, ya que su servidor se reiniciará cuando se realicen cambios en su proyecto de Flask.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

URL de enrutamiento

Con Flask, el enrutamiento de URL se realiza tradicionalmente con decoradores. Estos decoradores se pueden usar para enrutamiento estático, así como enrutamiento de URL con parámetros. Para el siguiente ejemplo, imagine que este script de Flask está ejecutando el sitio web `www.example.com` .

```
@app.route("/")
def index():
    return "You went to www.example.com"

@app.route("/about")
def about():
    return "You went to www.example.com/about"

@app.route("/users/guido-van-rossum")
    return "You went to www.example.com/guido-van-rossum"
```

Con esa última ruta, puede ver que dada una URL con / users / y el nombre del perfil, podríamos devolver un perfil. Como sería horriblemente ineficiente y desordenado incluir una `@app.route()` para cada usuario, Flask ofrece tomar parámetros de la URL:

```
@app.route("/users/<username>")
def profile(username):
    return "Welcome to the profile of " + username

cities = ["OMAHA", "MELBOURNE", "NEPAL", "STUTTGART", "LIMA", "CAIRO", "SHANGHAI"]

@app.route("/stores/locations/<city>")
def storefronts(city):
    if city in cities:
        return "Yes! We are located in " + city
    else:
        return "No. We are not located in " + city
```

Métodos HTTP

Los dos métodos HTTP más comunes son **GET** y **POST** . Flask puede ejecutar un código diferente de la misma URL dependiendo del método HTTP utilizado. Por ejemplo, en un servicio web con cuentas, es más conveniente enrutar la página de inicio de sesión y el proceso de inicio de sesión a través de la misma URL. Una solicitud GET, la misma que se realiza al abrir una URL en su navegador, debe mostrar el formulario de inicio de sesión, mientras que una solicitud POST (que lleva datos de inicio de sesión) debe procesarse por separado. También se crea una ruta

para manejar el método DELETE y PUT HTTP.

```
@app.route("/login", methods=["GET"])
def login_form():
    return "This is the login form"
@app.route("/login", methods=["POST"])
def login_auth():
    return "Processing your data"
@app.route("/login", methods=["DELETE", "PUT"])
def deny():
    return "This method is not allowed"
```

Para simplificar un poco el código, podemos importar el paquete de `request` desde el matraz.

```
from flask import request

@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Processing your data"
```

Para recuperar datos de la solicitud POST, debemos usar el paquete de `request` :

```
from flask import request
@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Username was " + request.form["username"] + " and password was " +
request.form["password"]
```

Archivos y plantillas

En lugar de escribir nuestro marcado HTML en las declaraciones de devolución, podemos usar la función `render_template()` :

```
from flask import Flask
from flask import render_template
app = Flask(__name__)

@app.route("/about")
def about():
    return render_template("about-us.html")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

Esto utilizará nuestro archivo de plantilla `about-us.html` . Para garantizar que nuestra aplicación

pueda encontrar este archivo, debemos organizar nuestro directorio en el siguiente formato:

```
- application.py
/templates
    - about-us.html
    - login-form.html
/static
    /styles
        - about-style.css
        - login-style.css
    /scripts
        - about-script.js
        - login-script.js
```

Lo más importante, las referencias a estos archivos en el HTML deben tener este aspecto:

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-
style.css')}}">
```

que dirigirá la aplicación para buscar `about-style.css` en la carpeta de estilos debajo de la carpeta estática. El mismo formato de ruta se aplica a todas las referencias a imágenes, estilos, scripts o archivos.

Jinja Templando

Al igual que Meteor.js, Flask se integra bien con los servicios de plantillas front-end. Frasco utiliza por defecto la plantilla de jinja. Las plantillas permiten que se utilicen pequeños fragmentos de código en el archivo HTML, como condicionales o bucles.

Cuando representamos una plantilla, todos los parámetros más allá del nombre del archivo de la plantilla se pasan al servicio de plantillas HTML. La siguiente ruta pasará el nombre de usuario y la fecha de ingreso (desde una función en otro lugar) al HTML.

```
@app.route("/users/<username>)
def profile(username):
    joinedDate = get_joined_date(username) # This function's code is irrelevant
    awards = get_awards(username) # This function's code is irrelevant
    # The joinDate is a string and awards is an array of strings
    return render_template("profile.html", username=username, joinDate=joinDate,
awards=awards)
```

Cuando esta plantilla se representa, puede usar las variables que se le pasan desde la función `render_template()` . Aquí están los contenidos de `profile.html` :

```
<!DOCTYPE html>
<html>
    <head>
        # if username
            <title>Profile of {{ username }}</title>
        # else
            <title>No User Found</title>
        # endif
    <head>
    <body>
```

```
        {% if username %}
            <h1>{{ username }} joined on the date {{ date }}</h1>
            {% if len(awards) > 0 %}
                <h3>{{ username }} has the following awards:</h3>
                <ul>
                {% for award in awards %}
                    <li>{{award}}</li>
                {% endfor %}
                </ul>
            {% else %}
                <h3>{{ username }} has no awards</h3>
            {% endif %}
        {% else %}
            <h1>No user was found under that username</h1>
        {% endif %}
        {# This is a comment and doesn't affect the output #}
    </body>
</html>
```

Los siguientes delimitadores se utilizan para diferentes interpretaciones:

- `{% ... %}` denota una declaración
- `{{ ... }}` denota una expresión donde se emite una plantilla
- `{# ... #}` denota un comentario (no incluido en la salida de la plantilla)
- `{# ... ##` implica que el resto de la línea debe interpretarse como una declaración

El objeto de solicitud

El objeto de `request` proporciona información sobre la solicitud que se realizó a la ruta. Para utilizar este objeto, debe importarse desde el módulo del matraz:

```
from flask import request
```

Parámetros de URL

En ejemplos anteriores se utilizaron `request.method` y `request.form` , sin embargo, también podemos usar la propiedad `request.args` para recuperar un diccionario de las claves / valores en los parámetros de la URL.

```
@app.route("/api/users/<username>")
def user_api(username):
    try:
        token = request.args.get("key")
        if key == "pA55w0Rd":
            if isUser(username): # The code of this method is irrelevant
                joined = joinDate(username) # The code of this method is irrelevant
                return "User " + username + " joined on " + joined
            else:
                return "User not found"
        else:
            return "Incorrect key"
    # If there is no key parameter
    except KeyError:
```

```
        return "No key provided"
```

Para autenticarse correctamente en este contexto, se necesitaría la siguiente URL (reemplazando el nombre de usuario con cualquier nombre de usuario):

```
www.example.com/api/users/guido-van-rossum?key=pa55w0Rd
```

Cargas de archivos

Si la carga de un archivo era parte del formulario enviado en una solicitud POST, los archivos se pueden manejar usando el objeto de `request` :

```
@app.route("/upload", methods=["POST"])
def upload_file():
    f = request.files["wordlist-upload"]
    f.save("/var/www/uploads/" + f.filename) # Store with the original filename
```

Galletas

La solicitud también puede incluir cookies en un diccionario similar a los parámetros de URL.

```
@app.route("/home")
def home():
    try:
        username = request.cookies.get("username")
        return "Your stored username is " + username
    except KeyError:
        return "No username cookies was found")
```

Lea Matraz en línea: https://riptutorial.com/es/python/topic/8682/matraz

Capítulo 121: Matrices multidimensionales

Examples

Listas en listas

Una buena manera de visualizar una matriz 2D es como una lista de listas. Algo como esto:

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

aquí la lista externa `lst` tiene tres cosas en él. cada una de esas cosas es otra lista: la primera es: `[1,2,3]` , la segunda es: `[4,5,6]` y la tercera es: `[7,8,9]` . Puede acceder a estas listas de la misma manera que accedería a otro elemento de una lista, como este:

```
print (lst[0])
#output: [1, 2, 3]

print (lst[1])
#output: [4, 5, 6]

print (lst[2])
#output: [7, 8, 9]
```

A continuación, puede acceder a los diferentes elementos en cada una de esas listas de la misma manera:

```
print (lst[0][0])
#output: 1

print (lst[0][1])
#output: 2
```

Aquí el primer número dentro de los corchetes `[]` significa obtener la lista en esa posición. En el ejemplo anterior, usamos el número `0` para obtener la lista en la posición 0, que es `[1,2,3]` . El segundo conjunto de `[]` corchetes significa obtener el elemento en esa posición de la lista interna. En este caso, utilizamos `0` y `1` la posición 0 en la lista que obtuvimos es el número `1` y en la 1ª posición es `2`

También puede establecer valores dentro de estas listas de la misma manera:

```
lst[0]=[10,11,12]
```

Ahora la lista es `[[10,11,12],[4,5,6],[7,8,9]]` . En este ejemplo, cambiamos toda la primera lista para que sea una lista completamente nueva.

```
lst[1][2]=15
```

Ahora la lista es `[[10,11,12],[4,5,15],[7,8,9]]` . En este ejemplo, cambiamos un solo elemento dentro de una de las listas internas. Primero ingresamos a la lista en la posición 1 y cambiamos el elemento dentro de la posición 2, que era 6 ahora es 15.

Listas en listas en listas en ...

Este comportamiento puede ser extendido. Aquí hay una matriz tridimensional:

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,313
```

Como probablemente sea obvio, esto se vuelve un poco difícil de leer. Usa barras invertidas para dividir las diferentes dimensiones:

```
[[[111,112,113],[121,122,123],[131,132,133]],\
 [[211,212,213],[221,222,223],[231,232,233]],\
 [[311,312,313],[321,322,323],[331,332,333]]]
```

Al anidar las listas como esta, puede extenderse a dimensiones arbitrariamente altas.

El acceso es similar a las matrices 2D:

```
print(myarray)
print(myarray[1])
print(myarray[2][1])
print(myarray[1][0][2])
etc.
```

Y la edición también es similar:

```
myarray[1]=new_n-1_d_list
myarray[2][1]=new_n-2_d_list
myarray[1][0][2]=new_n-3_d_list #or a single number if you're dealing with 3D arrays
etc.
```

Lea Matrices multidimensionales en línea: https://riptutorial.com/es/python/topic/8186/matrices-multidimensionales

Capítulo 122: Metaclases

Introducción

Las metaclases le permiten modificar profundamente el comportamiento de las clases de Python (en términos de cómo se definen, se crean instancias, se accede a ellas, y más) al reemplazar la metaclase de `type` que las nuevas clases usan de forma predeterminada.

Observaciones

Al diseñar su arquitectura, tenga en cuenta que muchas cosas que se pueden lograr con las metaclases también se pueden lograr utilizando una semántica más simple:

- La herencia tradicional es a menudo más que suficiente.
- Los decoradores de clase pueden combinar la funcionalidad en una clase con un enfoque ad hoc.
- Python 3.6 introduce `__init_subclass__()` que permite a una clase participar en la creación de su subclase.

Examples

Metaclases basicas

Cuando se llama a `type` con tres argumentos, se comporta como la clase (meta) que es y crea una nueva instancia, es decir. produce una nueva clase / tipo.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__               # <type 'type'>
Dummy().__class__.__class__   # <type 'type'>
```

Es posible subclase `type` para crear una metaclase personalizado.

```
class mytype(type):
    def __init__(cls, name, bases, dict):
        # call the base initializer
        type.__init__(cls, name, bases, dict)

        # perform custom initialization...
        cls.__custom_attribute__ = 2
```

Ahora, tenemos una nueva metaclase `mytype` personalizada que se puede utilizar para crear clases de la misma manera que el `type` .

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__               # <class '__main__.mytype'>
MyDummy().__class__.__class__   # <class '__main__.mytype'>
MyDummy.__custom_attribute__    # 2
```

Cuando creamos una nueva clase utilizando la palabra clave de `class` la metaclase se elige de forma predeterminada en función de las clases básicas.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

En el ejemplo anterior, la única clase de base es el `object` por lo que nuestra metaclase será el tipo de `object` , que es el `type` . Es posible anular el valor predeterminado, sin embargo, depende de si usamos Python 2 o Python 3:

Python 2.x 2.7

Un atributo de nivel de clase especial `__metaclass__` se puede usar para especificar la metaclase.

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy)  # <class '__main__.mytype'>
```

Python 3.x 3.0

Un argumento especial de palabra clave de `metaclass` especifica la metaclase.

```
class MyDummy(metaclass=mytype):
    pass
type(MyDummy)  # <class '__main__.mytype'>
```

Cualquier argumento de palabra clave (excepto `metaclass`) en la declaración de clase se pasará a la metaclase. Así, la `class MyDummy(metaclass=mytype, x=2)` pasará `x=2` como argumento de palabra clave al constructor `mytype` .

Lea esta descripción detallada de las meta-clases de python para más detalles.

Singletons utilizando metaclases

Un singleton es un patrón que restringe la creación de instancias de una clase a una instancia / objeto. Para más información sobre los patrones de diseño singleton de python, consulte aquí .

```
class SingletonType(type):
    def __call__(cls, *args, **kwargs):
        try:
            return cls.__instance
        except AttributeError:
            cls.__instance = super(SingletonType, cls).__call__(*args, **kwargs)
            return cls.__instance
```

Python 2.x 2.7

```
class MySingleton(object):
    __metaclass__ = SingletonType
```

Python 3.x 3.0

```
class MySingleton(metaclass=SingletonType):
    pass
```

```
MySingleton() is MySingleton()  # True, only one instantiation occurs
```

Usando una metaclase

Sintaxis de metaclase

Python 2.x 2.7

```
class MyClass(object):
    __metaclass__ = SomeMetaclass
```

Python 3.x 3.0

```
class MyClass(metaclass=SomeMetaclass):
    pass
```

Compatibilidad de Python 2 y 3 con six

```
import six

class MyClass(six.with_metaclass(SomeMetaclass)):
    pass
```

Funcionalidad personalizada con metaclases.

La funcionalidad de las metaclases se puede cambiar de modo que cada vez que se construye una clase, se imprime una cadena en la salida estándar o se lanza una excepción. Esta metaclase imprimirá el nombre de la clase que se está construyendo.

```
class VerboseMetaclass(type):

    def __new__(cls, class_name, class_parents, class_dict):
        print("Creating class ", class_name)
        new_class = super().__new__(cls, class_name, class_parents, class_dict)
        return new_class
```

Puedes usar la metaclase así:

```
class Spam(metaclass=VerboseMetaclass):
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

La salida estándar será:

```
Creating class Spam
[insert example string here]
```

Introducción a las metaclases

¿Qué es una metaclase?

En Python, todo es un objeto: enteros, cadenas, listas, incluso las funciones y las clases en sí son objetos. Y cada objeto es una instancia de una clase.

Para verificar la clase de un objeto x, se puede llamar al `type(x)` , entonces:

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

La mayoría de las clases en python son instancias de `type` . `type` sí mismo es también una clase. Tales clases cuyas instancias son también clases se llaman metaclases.

La metaclase mas simple

OK, entonces ya hay una metaclase en Python: `type` . ¿Podemos crear otro?

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

Eso no agrega ninguna funcionalidad, pero es una nueva metaclase, vea que MyClass ahora es una instancia de SimplestMetaclass:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

Una metaclase que hace algo

Una metaclase el que hace algo por lo general prevalece sobre `type` 's `__new__` , modificar algunas

propiedades de la clase que se creen, antes de llamar al original `__new__` que crea la clase:

```
class AnotherMetaclass(type):
    def __new__(cls, name, parents, dct):
        # cls is this class
        # name is the name of the class to be created
        # parents is the list of the class's parent classes
        # dct is the list of class's attributes (methods, static variables)

        # here all of the attributes can be modified before creating the class, e.g.

        dct['x'] = 8  # now the class will have a static variable x = 8

        # return value is the new class. super will take care of that
        return super(AnotherMetaclass, cls).__new__(cls, name, parents, dct)
```

La metaclase por defecto.

Es posible que hayas oído que todo en Python es un objeto. Es cierto, y todos los objetos tienen una clase:

```
>>> type(1)
int
```

El literal 1 es una instancia de `int` . Vamos a declarar una clase:

```
>>> class Foo(object):
...    pass
...
```

Ahora vamos a instanciarlo:

```
>>> bar = Foo()
```

¿Cuál es la clase de `bar` ?

```
>>> type(bar)
Foo
```

Agradable, el `bar` es una instancia de `Foo` . Pero, ¿cuál es la clase de `Foo` sí?

```
>>> type(Foo)
type
```

Ok, `Foo` sí es una instancia de `type` . ¿Qué tal el `type` sí mismo?

```
>>> type(type)
type
```

Entonces, ¿qué es una metaclase? Por ahora, simulemos que es solo un nombre elegante para la clase de una clase. Para llevar:

- Todo es un objeto en Python, así que todo tiene una clase.
- La clase de una clase se llama metaclase.
- La metaclase predeterminada es de `type` y, con mucho, es la metaclase más común.

Pero ¿por qué debería saber acerca de las metaclases? Bueno, Python en sí mismo es bastante "hackeable", y el concepto de metaclase es importante si estás haciendo cosas avanzadas como meta-programación o si quieres controlar cómo se inicializan tus clases.

Lea Metaclases en línea: https://riptutorial.com/es/python/topic/286/metaclases

Capítulo 123: Método Anulando

Examples

Método básico anulando

Este es un ejemplo de anulación básica en Python (por motivos de claridad y compatibilidad con Python 2 y 3, usando una nueva clase de estilo e `print` con `()`):

```
class Parent(object):
    def introduce(self):
        print("Hello!")

    def print_name(self):
        print("Parent")


class Child(Parent):
    def print_name(self):
        print("Child")


p = Parent()
c = Child()

p.introduce()
p.print_name()

c.introduce()
c.print_name()

$ python basic_override.py
Hello!
Parent
Hello!
Child
```

Cuando el `Child` se crea la clase, que hereda los métodos de la `Parent` clase. Esto significa que cualquier método que tenga la clase padre, la clase hija también tendrá. En el ejemplo, la `introduce` se define para la clase `Child` porque está definida para `Parent` , a pesar de no estar definida explícitamente en la definición de clase de `Child` .

En este ejemplo, la anulación se produce cuando `Child` define su propio método `print_name` . Si este método no fue declarado, entonces `c.print_name()` habría impreso `"Parent"` . Sin embargo, `Child` ha anulado el `Parent` definición 's de `print_name` , y por eso ahora al llamar `c.print_name()` , la palabra `"Child"` se imprime.

Lea Método Anulando en línea: https://riptutorial.com/es/python/topic/3131/metodo-anulando

Capítulo 124: Métodos de cuerda

Sintaxis

- str.capitalize () -> str
- str.casefold () -> str [solo para Python> 3.3]
- str.center (ancho [, fillchar]) -> str
- str.count (sub [, start [, end]]) -> int
- str.decode (encoding = "utf-8" [, errores]) -> unicode [solo en Python 2.x]
- str.encode (codificación = "utf-8", errores = "estricto") -> bytes
- str.endswith (sufijo [, inicio [, final]]) -> bool
- str.expandtabs (tabsize = 8) -> str
- str.find (sub [, start [, end]]) -> int
- str.format (* args, ** kwargs) -> str
- str.format_map (mapeo) -> str
- str.index (sub [, start [, end]]) -> int
- str.isalnum () -> bool
- str.isalpha () -> bool
- str.isdecimal () -> bool
- str.isdigit () -> bool
- identificador str. () -> bool
- str.islower () -> bool
- str.isnumeric () -> bool
- str.isprintable () -> bool
- str.isspace () -> bool
- str.istitle () -> bool
- str.isupper () -> bool
- str.join (iterable) -> str
- str.ljust (ancho [, fillchar]) -> str
- str.lower () -> str
- str.lstrip ([caracteres]) -> str
- str.maketrans estático (x [, y [, z]])
- str.partition (sep) -> (head, sep, tail)
- str.replace (antiguo, nuevo [, cuenta]) -> str
- str.rfind (sub [, start [, end]]) -> int
- str.rindex (sub [, inicio [, final]]) -> int
- str.rjust (ancho [, fillchar]) -> str
- str.rpartition (sep) -> (head, sep, tail)
- str.rsplit (sep = None, maxsplit = -1) -> lista de cadenas
- str.rstrip ([caracteres]) -> str
- str.split (sep = None, maxsplit = -1) -> lista de cadenas
- str.splitlines ([keepends]) -> lista de cadenas
- str.startswith (prefijo [, inicio [, final]]) -> libro
- str.strip ([caracteres]) -> str

- str.swapcase () -> str
- str.title () -> str
- str.translate (tabla) -> str
- str.upper () -> str
- str.zfill (ancho) -> str

Observaciones

Los objetos de cadena son inmutables, lo que significa que no se pueden modificar en su lugar como lo hace una lista. Debido a esto, los métodos en el tipo incorporado `str` siempre devuelven un **nuevo** objeto `str` , que contiene el resultado de la llamada al método.

Examples

Cambiar la capitalización de una cadena

El tipo de cadena de Python proporciona muchas funciones que actúan sobre la capitalización de una cadena. Éstos incluyen :

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

Con cadenas de Unicode (el valor predeterminado en Python 3), estas operaciones **no** son asignaciones 1: 1 o reversibles. La mayoría de estas operaciones están destinadas a fines de visualización, en lugar de a la normalización.

Python 3.x 3.3

`str.casefold()`

`str.casefold` crea una cadena en minúsculas que es adecuada para comparaciones que no distinguen entre mayúsculas y minúsculas. Esto es más agresivo que `str.lower` y puede modificar cadenas que ya están en minúsculas o hacer que las cadenas crezcan en longitud, y no está diseñada para fines de visualización.

```
"XßΣ".casefold()
# 'xssσ'

"XßΣ".lower()
# 'xßς'
```

Las transformaciones que tienen lugar en Casefolding están definidas por Unicode Consortium en el archivo CaseFolding.txt en su sitio web.

`str.upper()`

`str.upper` toma todos los caracteres de una cadena y los convierte a su equivalente en mayúsculas, por ejemplo:

```
"This is a 'string'.".upper()
# "THIS IS A 'STRING'."
```

`str.lower()`

`str.lower` hace lo contrario; toma todos los caracteres de una cadena y los convierte a su equivalente en minúsculas:

```
"This IS a 'string'.".lower()
# "this is a 'string'."
```

`str.capitalize()`

`str.capitalize` devuelve una versión en mayúscula de la cadena, es decir, hace que el primer carácter tenga mayúsculas y el resto sea inferior:

```
"this Is A 'String'.".capitalize() # Capitalizes the first character and lowercases all others
# "This is a 'string'."
```

`str.title()`

`str.title` devuelve el título de la versión de la cadena, es decir, todas las letras al principio de una palabra están en mayúsculas y las demás en minúsculas:

```
"this Is a 'String'".title()
# "This Is A 'String'"
```

`str.swapcase()`

`str.swapcase` devuelve un nuevo objeto de cadena en el que todos los caracteres en minúsculas se cambian a mayúsculas y todos los caracteres en mayúsculas a inferiores:

```
"this iS A STRiNG".swapcase() #Swaps case of each character
# "THIS Is a strIng"
```

Uso como métodos de clase `str`

Vale la pena señalar que estos métodos pueden llamarse en objetos de cadena (como se muestra arriba) o como un método de clase de la clase `str` (con una llamada explícita a `str.upper` , etc.)

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

Esto es más útil cuando se aplica uno de estos métodos a muchas cadenas a la vez, por ejemplo, una función de `map` .

```
map(str.upper,["These","are","some","'strings'"])
# ['THESE', 'ARE', 'SOME', "'STRINGS'"]
```

Dividir una cadena basada en un delimitador en una lista de cadenas

`str.split(sep=None, maxsplit=-1)`

`str.split` toma una cadena y devuelve una lista de subcadenas de la cadena original. El comportamiento difiere dependiendo de si el argumento `sep` se proporciona u omite.

Si no se proporciona `sep` , o es `None` , entonces la división tiene lugar donde hay espacios en blanco. Sin embargo, los espacios en blanco iniciales y finales se ignoran, y varios caracteres de espacios en blanco consecutivos se tratan igual que un solo espacio en blanco:

```
>>> "This is a sentence.".split()
['This', 'is', 'a', 'sentence.']

>>> " This is    a sentence.  ".split()
['This', 'is', 'a', 'sentence.']

>>> "            ".split()
[]
```

El parámetro `sep` se puede usar para definir una cadena delimitadora. La cadena original se divide donde se produce la cadena delimitadora, y el propio delimitador se descarta. Varios delimitadores consecutivos *no se* tratan de la misma manera que una sola ocurrencia, sino que hacen que se creen cadenas vacías.

```
>>> "This is a sentence.".split(' ')
['This', 'is', 'a', 'sentence.']

>>> "Earth,Stars,Sun,Moon".split(',')
['Earth', 'Stars', 'Sun', 'Moon']

>>> " This is    a sentence.  ".split(' ')
['', 'This', 'is', '', '', '', 'a', 'sentence.', '', '']

>>> "This is a sentence.".split('e')
['This is a s', 'nt', 'nc', '.']

>>> "This is a sentence.".split('en')
['This is a s', 't', 'ce.']
```

El valor predeterminado es dividir en *cada* aparición del delimitador, sin embargo, el parámetro `maxsplit` limita el número de divisiones que se producen. El valor predeterminado de `-1` significa que no hay límite:

```
>>> "This is a sentence.".split('e', maxsplit=0)
['This is a sentence.']

>>> "This is a sentence.".split('e', maxsplit=1)
['This is a s', 'ntence.']

>>> "This is a sentence.".split('e', maxsplit=2)
['This is a s', 'nt', 'nce.']

>>> "This is a sentence.".split('e', maxsplit=-1)
['This is a s', 'nt', 'nc', '.']
```

`str.rsplit(sep=None, maxsplit=-1)`

`str.rsplit` ("división derecha") difiere de `str.split` ("división izquierda") cuando se especifica `maxsplit` . La división comienza al final de la cadena en lugar de al principio:

```
>>> "This is a sentence.".rsplit('e', maxsplit=1)
['This is a sentenc', '.']

>>> "This is a sentence.".rsplit('e', maxsplit=2)
['This is a sent', 'nc', '.']
```

Nota : Python especifica el número máximo de *divisiones* realizadas, mientras que la mayoría de los otros lenguajes de programación especifican el número máximo de *subcadenas* creadas. Esto puede crear confusión al portar o comparar código.

Reemplace todas las ocurrencias de una subcadena por otra subcadena

El tipo `str` de Python también tiene un método para reemplazar las ocurrencias de una subcadena con otra subcadena en una cadena dada. Para casos más exigentes, uno puede usar `re.sub` .

`str.replace(old, new[, count])` :

`str.replace` toma dos argumentos `old` y `new` que contiene el `old` sub-secuencia que va a ser reemplazado por el `new` sub-secuencia. El `count` argumentos opcional especifica el número de reemplazos que se realizarán:

Por ejemplo, para reemplazar `'foo'` con `'spam'` en la siguiente cadena, podemos llamar a `str.replace` con `old = 'foo'` y `new = 'spam'` :

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
"Make sure to spam your sentence."
```

Si la cadena dada contiene múltiples ejemplos que coinciden con el `old` argumento, **todas las** apariciones se reemplazan por el valor suministrado en `new` :

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
"It can spam multiple examples of spam if you want."
```

A menos que, por supuesto, proporcionemos un valor para el `count` . En este caso, las ocurrencias de `count` serán reemplazadas:

```
>>> """It can foo multiple examples of foo if you want, \
... or you can limit the foo with the third argument.""".replace('foo', 'spam', 1)
'It can spam multiple examples of foo if you want, or you can limit the foo with the third
argument.'
```

str.format y f-strings: formatea valores en una cadena

Python proporciona interpolación de cadenas y funcionalidad de formato a través de la función `str.format` , introducida en la versión 2.6 y f-cadenas introducidas en la versión 3.6.

Dadas las siguientes variables:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

Las siguientes afirmaciones son todas equivalentes

```
"10 1.5 foo ['a', 1, 2] {'a': 1, 2: 'foo'}"
```

```
>>> "{} {} {} {} {}".format(i, f, s, l, d)

>>> str.format("{} {} {} {} {}", i, f, s, l, d)

>>> "{0} {1} {2} {3} {4}".format(i, f, s, l, d)

>>> "{0:d} {1:0.1f} {2} {3!r} {4!r}".format(i, f, s, l, d)

>>> "{i:d} {f:0.1f} {s} {l!r} {d!r}".format(i=i, f=f, s=s, l=l, d=d)
```

```
>>> f"{i} {f} {s} {l} {d}"

>>> f"{i:d} {f:0.1f} {s} {l!r} {d!r}"
```

Para referencia, Python también admite calificadores de estilo C para el formato de cadenas. Los ejemplos a continuación son equivalentes a los anteriores, pero se prefieren las versiones de `str.format` debido a los beneficios en flexibilidad, consistencia de notación y extensibilidad:

```
"%d %0.1f %s %r %r" % (i, f, s, l, d)

"%(i)d %(f)0.1f %(s)s %(l)r %(d)r" % dict(i=i, f=f, s=s, l=l, d=d)
```

Las llaves que se usan para la interpolación en `str.format` de `str.format` también se pueden numerar para reducir la duplicación al formatear cadenas. Por ejemplo, los siguientes son equivalentes:

```
"I am from Australia. I love cupcakes from Australia!"
```

```
>>> "I am from {}. I love cupcakes from {}!".format("Australia", "Australia")

>>> "I am from {0}. I love cupcakes from {0}!".format("Australia")
```

Si bien la documentación oficial de Python es, como es habitual, lo suficientemente exhaustiva, pyformat.info tiene un gran conjunto de ejemplos con explicaciones detalladas.

Además, los caracteres `{` y `}` pueden escaparse utilizando corchetes dobles:

```
"{'a': 5, 'b': 6}"
```

```
>>> "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)

>>> f"{{'{'a'}': {5}, '{'b'}': {6}}"
```

Consulte Formato de cadena para obtener información adicional. `str.format()` se propuso en PEP 3101 y f-strings en PEP 498 .

Contando el número de veces que una subcadena aparece en una cadena

Hay un método disponible para contar el número de apariciones de una `str.count` en otra cadena, `str.count` .

```
str.count(sub[, start[, end]])
```

`str.count` devuelve un `int` indica el número de apariciones no superpuestas de la `str.count` `sub` en otra cadena. Los argumentos opcionales `start` y `end` indican el principio y el final en el que se realizará la búsqueda. De forma predeterminada, `start = 0` y `end = len(str)` significa que se buscará en toda la cadena:

```
>>> s = "She sells seashells by the seashore."
>>> s.count("sh")
2
>>> s.count("se")
3
>>> s.count("sea")
2
>>> s.count("seashells")
1
```

Al especificar un valor diferente para el `start` , al `end` , podemos obtener una búsqueda más localizada y contar, por ejemplo, si el `start` es igual a `13` la llamada a:

```
>>> s.count("sea", start)
1
```

es equivalente a:

```
>>> t = s[start:]
>>> t.count("sea")
1
```

Prueba los caracteres iniciales y finales de una cadena.

Para probar el principio y el final de una cadena dada en Python, uno puede usar los métodos `str.startswith()` y `str.endswith()` .

`str.startswith(prefix[, start[, end]])`

Como su nombre lo indica, `str.startswith` se usa para probar si una cadena dada comienza con los caracteres dados en el `prefix` .

```
>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False
```

Los argumentos opcionales de `start` y `end` especifican los puntos de inicio y finalización a partir de los cuales se iniciarán y finalizarán las pruebas. En el siguiente ejemplo, al especificar un valor de inicio de `2` , se buscará nuestra cadena desde la posición `2` y después:

```
>>> s.startswith("is", 2)
True
```

Esto da como resultado `True` desde `s[2] == 'i'` y `s[3] == 's'` .

También puede usar una `tuple` para verificar si comienza con alguna de un conjunto de cadenas

```
>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False
```

`str.endswith(prefix[, start[, end]])`

`str.endswith` es exactamente similar a `str.startswith` la única diferencia es que busca caracteres finales y no caracteres iniciales. Por ejemplo, para probar si una cadena termina en una parada completa, se podría escribir:

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

al igual que con `startswith` se pueden usar más de un carácter como secuencia final:

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

También puede usar una `tuple` para verificar si termina con cualquiera de un conjunto de cadenas

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

Probando de qué está compuesta una cuerda

El tipo `str` de Python también presenta una serie de métodos que se pueden usar para evaluar el contenido de una cadena. Estos son `str.isalpha` , `str.isdigit` , `str.isalnum` , `str.isspace` . La capitalización se puede probar con `str.isupper` , `str.islower` y `str.istitle` .

`str.isalpha`

`str.isalpha` no toma argumentos y devuelve `True` si todos los caracteres de una cadena dada son alfabéticos, por ejemplo:

```
>>> "Hello World".isalpha()  # contains a space
False
>>> "Hello2World".isalpha()  # contains a number
False
>>> "HelloWorld!".isalpha()  # contains punctuation
False
>>> "HelloWorld".isalpha()
True
```

Como un caso de borde, la cadena vacía se evalúa como `False` cuando se usa con `"".isalpha()` .

`str.isupper` , `str.islower` , `str.istitle`

Estos métodos prueban el uso de mayúsculas en una cadena dada.

`str.isupper` es un método que devuelve `True` si todos los caracteres de una cadena dada están en mayúsculas y `False` caso contrario.

```
>>> "HeLLO WORLD".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> "".isupper()
False
```

A la inversa, `str.islower` es un método que devuelve `True` si todos los caracteres de una cadena dada son minúsculas y `False` contrario.

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` devuelve `True` si la cadena dada es un título cargado; es decir, cada palabra comienza con un carácter en mayúscula seguido de caracteres en minúscula.

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

`str.isdecimal` , `str.isdigit` , `str.isnumeric`

`str.isdecimal` devuelve si la cadena es una secuencia de dígitos decimales, adecuada para representar un número decimal.

`str.isdigit` incluye dígitos que no están en una forma adecuada para representar un número decimal, como los dígitos en superíndice.

`str.isnumeric` incluye cualquier valor numérico, incluso si no son dígitos, como valores fuera del rango 0-9.

```
            isdecimal    isdigit   isnumeric

12345         True         True        True
□2□□5         True         True        True
①²³□₅       False        True        True
⑩□            False        False       True
Five          False        False       False
```

Bytestrings (`bytes` en Python 3, `str` en Python 2), solo admite `isdigit` , que solo verifica los dígitos ASCII básicos.

Al igual que con `str.isalpha` , la cadena vacía se evalúa como `False` .

`str.isalnum`

Esta es una combinación de `str.isalpha` y `str.isnumeric` , específicamente se evalúa como `True` si todos los caracteres en la cadena dada son **alfanuméricos** , es decir, consisten en caracteres

alfabéticos *o* numéricos:

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
>>> "Hello World".isalnum()  # contains whitespace
False
```

---

str.isspace

Se evalúa como `True` si la cadena solo contiene caracteres de espacio en blanco.

```
>>> "\t\r\n".isspace()
True
>>> " ".isspace()
True
```

A veces, una cadena se ve "vacía" pero no sabemos si es porque solo contiene espacios en blanco o ningún carácter.

```
>>> "".isspace()
False
```

Para cubrir este caso necesitamos una prueba adicional.

```
>>> my_str = ''
>>> my_str.isspace()
False
>>> my_str.isspace() or not my_str
True
```

Pero la forma más corta de probar si una cadena está vacía o simplemente contiene caracteres de espacio en blanco es usar la `strip` (sin argumentos elimina todos los caracteres de espacio en blanco iniciales y finales)

```
>>> not my_str.strip()
True
```

str.translate: Traducir caracteres en una cadena

Python admite un método de `translate` en el tipo `str` que le permite especificar la tabla de traducción (utilizada para los reemplazos), así como cualquier carácter que deba eliminarse en el proceso.

```
str.translate(table[, deletechars])
```

Parámetro	Descripción
`table`	Es una tabla de búsqueda que define la asignación de un carácter a otro.
`deletechars`	Una lista de caracteres que se eliminarán de la cadena.

El método `maketrans` (`str.maketrans` en Python 3 y `string.maketrans` en Python 2) le permite generar una tabla de traducción.

```
>>> translation_table = str.maketrans("aeiou", "12345")
>>> my_string = "This is a string!"
>>> translated = my_string.translate(translation_table)
'Th3s 3s 1 str3ng!'
```

El método de `translate` devuelve una cadena que es una copia traducida de la cadena original.

Puede establecer el argumento de la `table` en `None` si solo necesita eliminar caracteres.

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
'ths syntx s vry sfl'
```

Eliminar caracteres iniciales / finales no deseados de una cadena

Se proporcionan tres métodos que ofrecen la posibilidad de eliminar los caracteres `str.strip` y finales de una cadena: `str.strip` , `str.rstrip` y `str.lstrip` . Los tres métodos tienen la misma firma y los tres devuelven un nuevo objeto de cadena con caracteres no deseados eliminados.

`str.strip([chars])`

`str.strip` actúa sobre una cadena dada y elimina (elimina) cualquier carácter `str.strip` o final contenido en los `chars` argumento; Si no se suministran `chars` o es `None` , todos los caracteres de espacios en blanco se eliminan de forma predeterminada. Por ejemplo:

```
>>> "    a line with leading and trailing space     ".strip()
'a line with leading and trailing space'
```

Si se proporcionan `chars` , todos los caracteres contenidos en él se eliminan de la cadena, que se devuelve. Por ejemplo:

```
>>> ">>> a Python prompt".strip('> ')  # strips '>' character and space character
'a Python prompt'
```

`str.rstrip([chars])` y `str.lstrip([chars])`

Estos métodos tienen una semántica y argumentos similares con `str.strip()` , su diferencia radica en la dirección desde la que comienzan. `str.rstrip()` comienza desde el final de la cadena,

mientras que `str.lstrip()` divide desde el principio de la cadena.

Por ejemplo, usando `str.rstrip` :

```
>>> "     spacious string      ".rstrip()
'     spacious string'
```

Mientras, usando `str.lstrip` :

```
>>> "     spacious string      ".rstrip()
'spacious string      '
```

Comparaciones de cadenas insensibles al caso

La comparación de cadenas en una forma que no distingue entre mayúsculas y minúsculas parece algo trivial, pero no lo es. Esta sección solo considera las cadenas Unicode (la predeterminada en Python 3). Tenga en cuenta que Python 2 puede tener debilidades sutiles en relación con Python 3; el manejo de Unicode de este último es mucho más completo.

Lo primero que hay que tener en cuenta es que las conversiones de eliminación de casos en Unicode no son triviales. Hay texto para el que `text.lower() != text.upper().lower()` , como `"ß"` :

```
>>> "ß".lower()
'ß'

>>> "ß".upper().lower()
'ss'
```

Pero digamos que quería comparar `"BUSSE"` y `"Buße"` . Demonios, es probable que también quieras comparar `"BUSSE"` y `"BU□E"` iguales, esa es la forma de capital más nueva. La forma recomendada es usar `casefold` :

Python 3.x 3.3

```
>>> help(str.casefold)
"""
Help on method_descriptor:

casefold(...)
      S.casefold() -> str

     Return a version of S suitable for caseless comparisons.
"""
```

No se limite a usar `lower` . Si la `casefold` no está disponible, hacer `.upper().lower()` ayuda (pero solo un poco).

Entonces deberías considerar los acentos. Si su procesador de fuentes es bueno, probablemente piense que `"ê" == "ê "` , pero no lo hace:

```
>>> "ê" == "ê "
```

```
False
```

Esto es porque en realidad son

```
>>> import unicodedata

>>> [unicodedata.name(char) for char in "ê"]
['LATIN SMALL LETTER E WITH CIRCUMFLEX']

>>> [unicodedata.name(char) for char in "ê "]
['LATIN SMALL LETTER E', 'COMBINING CIRCUMFLEX ACCENT']
```

La forma más sencilla de lidiar con esto es `unicodedata.normalize` . Probablemente desee utilizar la normalización **NFKD** , pero no dude en consultar la documentación. Entonces uno hace

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê ")
True
```

Para finalizar, aquí esto se expresa en funciones:

```
import unicodedata

def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

Unir una lista de cadenas en una cadena

Una cadena puede usarse como separador para unir una lista de cadenas en una sola cadena usando el método `join()` . Por ejemplo, puede crear una cadena en la que cada elemento de una lista esté separado por un espacio.

```
>>> " ".join(["once","upon","a","time"])
"once upon a time"
```

El siguiente ejemplo separa los elementos de cadena con tres guiones.

```
>>> "---".join(["once", "upon", "a", "time"])
"once---upon---a---time"
```

Constantes útiles del módulo de cadena

El módulo de `string` de Python proporciona constantes para operaciones relacionadas con cadenas. Para usarlos, importa el módulo de `string` :

```
>>> import string
```

string.ascii_letters :

Concatenación de `ascii_lowercase` y `ascii_uppercase` :

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

---

string.ascii_lowercase

Contiene todos los caracteres ASCII en minúsculas:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

---

string.ascii_uppercase :

Contiene todos los caracteres ASCII en mayúsculas:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

---

string.digits :

Contiene todos los caracteres de dígitos decimales:

```
>>> string.digits
'0123456789'
```

---

string.hexdigits :

Contiene todos los caracteres de dígitos hexadecimales:

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

---

string.octaldigits :

Contiene todos los caracteres de dígitos octales:

```
>>> string.octaldigits
'01234567'
```

---

string.punctuation :

Contiene todos los caracteres que se consideran puntuación en la configuración regional de `C` :

```
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

`string.whitespace` :

Contiene todos los caracteres ASCII considerados espacios en blanco:

```
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

En el modo de script, `print(string.whitespace)` imprimirá los caracteres reales, use `str` para obtener la cadena devuelta arriba.

`string.printable` :

Contiene todos los caracteres que se consideran imprimibles; una combinación de `string.digits` , `string.ascii_letters` , `string.punctuation` y `string.whitespace` .

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-
./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Invertir una cadena

Una cadena puede revertirse usando la función `reversed()` incorporada, que toma una cadena y devuelve un iterador en orden inverso.

```
>>> reversed('hello')
<reversed object at 0x0000000000000000>
>>> [char for char in reversed('hello')]
['o', 'l', 'l', 'e', 'h']
```

`reversed()` puede envolverse en una llamada a `''.join()` para crear una cadena desde el iterador.

```
>>> ''.join(reversed('hello'))
'olleh'
```

Si bien el uso de `reversed()` puede ser más legible para los usuarios no iniciados de Python, el uso del corte extendido con un paso de `-1` es más rápido y conciso. Aquí, intente implementarlo como función:

```
>>> def reversed_string(main_string):
...     return main_string[::-1]
...
>>> reversed_string('hello')
```

```
'olleh'
```

Justificar cuerdas

Python proporciona funciones para justificar cadenas, permitiendo el relleno de texto para que la alineación de varias cadenas sea mucho más fácil.

A continuación se muestra un ejemplo de `str.ljust` y `str.rjust` :

```
interstates_lengths = {
    5: (1381, 2222),
    19: (63, 102),
    40: (2555, 4112),
    93: (189,305),
}
for road, length in interstates_lengths.items():
    miles,kms = length
    print('{} -> {} mi. ({} km.)'.format(str(road).rjust(4), str(miles).ljust(4),
str(kms).ljust(4)))
```

```
  40 -> 2555 mi. (4112 km.)
  19 -> 63   mi. (102  km.)
   5 -> 1381 mi. (2222 km.)
  93 -> 189  mi. (305  km.)
```

`ljust` y `rjust` son muy similares. Ambos tienen un parámetro de `width` y un parámetro de `fillchar` opcional. Cualquier cadena creada por estas funciones es al menos tan larga como el parámetro de `width` que se pasó a la función. Si la cadena es más larga que el `width` alread, no se trunca. El argumento `fillchar` , que por defecto es el carácter de espacio `' '` debe ser un solo carácter, no una cadena de caracteres múltiples.

La función `ljust` el final de la cadena con la que se llama con el `fillchar` hasta que tenga una `width` caracteres. La función `rjust` el principio de la cadena de una manera similar. Por lo tanto, `l` y `r` en los nombres de estas funciones se refieren al lado en el que la cadena original, *no el* `fillchar` , está posicionado en la cadena de salida.

Conversión entre str o bytes de datos y caracteres Unicode

El contenido de los archivos y mensajes de la red puede representar caracteres codificados. A menudo necesitan ser convertidos a Unicode para una visualización adecuada.

En Python 2, es posible que necesite convertir datos de str a caracteres Unicode. El valor predeterminado (`''` , `""` , etc.) es una cadena ASCII, con cualquier valor fuera del rango ASCII mostrado como valores escapados. Las cadenas Unicode son `u''` (o `u""` , etc.).

Python 2.x 2.3

```
# You get "© abc" encoded in UTF-8 from a file, network, or other data source

s = '\xc2\xa9 abc'  # s is a byte array, not a string of characters
                    # Doesn't know the original was UTF-8
```

```
                    # Default form of string literals in Python 2
s[0]                # '\xc2' - meaningless byte (without context such as an encoding)
type(s)             # str - even though it's not a useful one w/o having a known encoding

u = s.decode('utf-8')  # u'\xa9 abc'
                       # Now we have a Unicode string, which can be read as UTF-8 and printed
properly
                       # In Python 2, Unicode string literals need a leading u
                       # str.decode converts a string which may contain escaped bytes to a
Unicode string
u[0]                # u'\xa9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # unicode

u.encode('utf-8')   # '\xc2\xa9 abc'
                    # unicode.encode produces a string with escaped bytes for non-ASCII
characters
```

En Python 3 es posible que necesite convertir matrices de bytes (denominadas 'literal de byte') a cadenas de caracteres Unicode. El valor predeterminado es ahora una cadena Unicode, y los literales de bytestring ahora deben ingresarse como `b''` , `b""` , etc. Un literal de byte devolverá `True` a `isinstance(some_val, byte)` , asumiendo que `some_val` es una cadena que podría estar codificada como bytes.

Python 3.x 3.0

```
# You get from file or network "© abc" encoded in UTF-8

s = b'\xc2\xa9 abc' # s is a byte array, not characters
                    # In Python 3, the default string literal is Unicode; byte array literals
need a leading b
s[0]                # b'\xc2' - meaningless byte (without context such as an encoding)
type(s)             # bytes - now that byte arrays are explicit, Python can show that.

u = s.decode('utf-8')  # '© abc' on a Unicode terminal
                       # bytes.decode converts a byte array to a string (which will, in Python
3, be Unicode)
u[0]                # '\u00a9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # str
                    # The default string literal in Python 3 is UTF-8 Unicode

u.encode('utf-8')   # b'\xc2\xa9 abc'
                    # str.encode produces a byte array, showing ASCII-range bytes as unescaped
characters.
```

Cadena contiene

Python hace que sea extremadamente intuitivo comprobar si una cadena contiene una subcadena dada. Solo usa el operador `in` :

```
>>> "foo" in "foo.baz.bar"
True
```

Nota: probar una cadena vacía siempre resultará en `True` :

```
>>> "" in "test"
```

```
True
```

Lea Métodos de cuerda en línea: https://riptutorial.com/es/python/topic/278/metodos-de-cuerda

Capítulo 125: Métodos definidos por el usuario

Examples

Creando objetos de método definidos por el usuario

Los objetos de método definidos por el usuario pueden crearse cuando se obtiene un atributo de una clase (quizás a través de una instancia de esa clase), si ese atributo es un objeto de función definido por el usuario, un objeto de método definido por el usuario no vinculado o un objeto de método de clase.

```
class A(object):
    # func: A user-defined function object
    #
    # Note that func is a function object when it's defined,
    # and an unbound method object when it's retrieved.
    def func(self):
        pass

    # classMethod: A class method
    @classmethod
    def classMethod(self):
        pass

class B(object):
    # unboundMeth: A unbound user-defined method object
    #
    # Parent.func is an unbound user-defined method object here,
    # because it's retrieved.
    unboundMeth = A.func

a = A()
b = B()

print A.func
# output: <unbound method A.func>
print a.func
# output: <bound method A.func of <__main__.A object at 0x10e9ab910>>
print B.unboundMeth
# output: <unbound method A.func>
print b.unboundMeth
# output: <unbound method A.func>
print A.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
print a.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
```

Cuando el atributo es un objeto de método definido por el usuario, un nuevo objeto de método solo se crea si la clase de la cual se está recuperando es la misma clase que la clase almacenada en el objeto de método original, o una clase derivada de la misma; de lo contrario, el objeto del método original se usa tal como es.

```
# Parent: The class stored in the original method object
class Parent(object):
    # func: The underlying function of original method object
    def func(self):
        pass
    func2 = func

# Child: A derived class of Parent
class Child(Parent):
    func = Parent.func

# AnotherClass: A different class, neither subclasses nor subclassed
class AnotherClass(object):
    func = Parent.func

print Parent.func is Parent.func                # False, new object created
print Parent.func2 is Parent.func2              # False, new object created
print Child.func is Child.func                  # False, new object created
print AnotherClass.func is AnotherClass.func    # True, original object used
```

Ejemplo de tortuga

El siguiente es un ejemplo del uso de una función definida por el usuario para ser llamada varias veces (∞) en un script con facilidad.

```
import turtle, time, random #tell python we need 3 different modules
turtle.speed(0) #set draw speed to the fastest
turtle.colormode(255) #special colormode
turtle.pensize(4) #size of the lines that will be drawn
def triangle(size): #This is our own function, in the parenthesis is a variable we have
defined that will be used in THIS FUNCTION ONLY. This fucntion creates a right triangle
    turtle.forward(size) #to begin this function we go forward, the amount to go forward by is
the variable size
    turtle.right(90) #turn right by 90 degree
    turtle.forward(size) #go forward, again with variable
    turtle.right(135) #turn right again
    turtle.forward(size * 1.5) #close the triangle. thanks to the Pythagorean theorem we know
that this line must be 1.5 times longer than the other two(if they are equal)
while(1): #INFINITE LOOP
    turtle.setpos(random.randint(-200, 200), random.randint(-200, 200)) #set the draw point to
a random (x,y) position
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize the RGB color
    triangle(random.randint(5, 55)) #use our function, because it has only one variable we can
simply put a value in the parenthesis. The value that will be sent will be random between 5 -
55, end the end it really just changes ow big the triangle is.
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize color again
```

Lea Métodos definidos por el usuario en línea:
https://riptutorial.com/es/python/topic/3965/metodos-definidos-por-el-usuario

Capítulo 126: Mixins

Sintaxis

- class *ClassName* (*MainClass* , *Mixin1* , *Mixin2* , ...): # Se utiliza para declarar una clase con el nombre *ClassName* , main (first) class *MainClass* y mixins *Mixin1* , *Mixin2* , etc.
- class *ClassName* (*Mixin1* , *MainClass* , *Mixin2* , ...): # La clase 'main' no tiene que ser la primera clase; Realmente no hay diferencia entre esto y la mezcla

Observaciones

Agregar un mixin a una clase se parece mucho a agregar una superclase, porque es más o menos eso. Un objeto de una clase con la mezcla *Foo* también será una instancia de *Foo* , e `isinstance(instance, Foo)` devolverá verdadero

Examples

Mezclar

Un **Mixin** es un conjunto de propiedades y métodos que se pueden usar en diferentes clases, que *no* provienen de una clase base. En los lenguajes de programación orientada a objetos, normalmente se usa la *herencia* para dar a los objetos de diferentes clases la misma funcionalidad; si un conjunto de objetos tiene alguna habilidad, pones esa habilidad en una clase base de la cual ambos objetos *heredan* .

> Por ejemplo, supongamos que tiene las clases `Car` , `Boat` y `Plane` . Los objetos de todas estas clases tienen la capacidad de viajar, por lo que obtienen la función de `travel` . En este escenario, todos viajan de la misma manera básica, también; consiguiendo una ruta, y moviéndose a lo largo de ella. Para implementar esta función, podría derivar todas las clases de `Vehicle` y poner la función en esa clase compartida:

```
class Vehicle(object):
   """A generic vehicle class."""

   def __init__(self, position):
       self.position = position

   def travel(self, destination):
       route = calculate_route(from=self.position, to=destination)
       self.move_along(route)

class Car(Vehicle):
   ...

class Boat(Vehicle):
   ...

class Plane(Vehicle):
```

```
    ...
```

Con este código, puede llamar a `travel` en un automóvil (`car.travel("Montana")`), barco (`boat.travel("Hawaii")`), y avión (`plane.travel("France")`)

Sin embargo, ¿qué sucede si tiene una funcionalidad que no está disponible para una clase base? Digamos, por ejemplo, que quieres darle a `Car` una radio y la posibilidad de usarla para reproducir una canción en una estación de radio, con `play_song_on_station` , pero también tienes un `Clock` que puede usar una radio. `Car` y `Clock` podrían compartir una clase base (`Machine`). Sin embargo, no todas las máquinas pueden reproducir canciones; `Boat` y `Plane` no pueden (al menos en este ejemplo). Entonces, ¿cómo lograr sin duplicar el código? Puedes usar un mixin. En Python, dar una mezcla a una clase es tan simple como agregarla a la lista de subclases, como esto

```
class Foo(main_super, mixin): ...
```

`Foo` heredará todas las propiedades y métodos de `main_super` , pero también los de `mixin` .

Por lo tanto, para dar a las clases `Car` y reloj la posibilidad de usar una radio, puede anular `Car` del último ejemplo y escribir esto:

```
class RadioUserMixin(object):
    def __init__(self):
        self.radio = Radio()

    def play_song_on_station(self, station):
        self.radio.set_station(station)
        self.radio.play_song()

class Car(Vehicle, RadioUserMixin):
    ...

class Clock(Vehicle, RadioUserMixin):
    ...
```

Ahora puedes llamar a `car.play_song_on_station(98.7)` y `clock.play_song_on_station(101.3)` , pero no a algo como `boat.play_song_on_station(100.5)`

Lo importante con los mixins es que le permiten agregar funcionalidad a objetos muy diferentes, que no comparten una subclase "principal" con esta funcionalidad, pero aún así comparten el código para ello. Sin los mixins, hacer algo como el ejemplo anterior sería mucho más difícil y / o podría requerir alguna repetición.

Métodos de anulación en Mixins

Los mixins son una clase de clase que se usa para "mezclar" propiedades y métodos adicionales en una clase. Por lo general, esto está bien porque muchas veces las clases mixtas no se anulan entre sí, o los métodos de la clase base. Pero si anula métodos o propiedades en sus combinaciones, esto puede llevar a resultados inesperados porque en Python la jerarquía de

clases se define de derecha a izquierda.

Por ejemplo, tome las siguientes clases

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class BaseClass(object):
    def test(self):
        print "Base"

class MyClass(BaseClass, Mixin1, Mixin2):
    pass
```

En este caso, la clase Mixin2 es la clase base, extendida por Mixin1 y finalmente por BaseClass. Por lo tanto, si ejecutamos el siguiente fragmento de código:

```
>>> x = MyClass()
>>> x.test()
Base
```

Vemos que el resultado devuelto es de la clase Base. Esto puede provocar errores inesperados en la lógica de su código y debe tenerse en cuenta y tenerse en cuenta.

Lea Mixins en línea: https://riptutorial.com/es/python/topic/4359/mixins

Capítulo 127: Modismos

Examples

Inicializaciones clave del diccionario

Prefiera el método `dict.get` si no está seguro si la clave está presente. Le permite devolver un valor predeterminado si no se encuentra la clave. El método tradicional `dict[key]` generaría una excepción `KeyError` .

En lugar de hacer

```
def add_student():
    try:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

Hacer

```
def add_student():
        students['count'] = students.get('count', 0) + 1
```

Variables de conmutacion

Para cambiar el valor de dos variables puede usar el desempaquetado de la tupla.

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

Use la prueba de valor de verdad

Python convertirá implícitamente cualquier objeto a un valor booleano para la prueba, así que úsalo siempre que sea posible.

```
# Good examples, using implicit truth testing
if attr:
    # do something

if not attr:
    # do something

# Bad examples, using specific types
if attr == 1:
```

```
    # do something

if attr == True:
    # do something

if attr != '':
    # do something

# If you are looking to specifically check for None, use 'is' or 'is not'
if attr is None:
    # do something
```

Esto generalmente produce un código más legible, y generalmente es mucho más seguro cuando se trata de tipos inesperados.

Haga clic aquí para obtener una lista de lo que se evaluará como `False` .

Prueba de "__main__" para evitar la ejecución inesperada del código

Es una buena práctica probar la variable `__name__` del programa que llama antes de ejecutar su código.

```
import sys

def main():
    # Your code starts here

    # Don't forget to provide a return code
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

El uso de este patrón asegura que su código solo se ejecute cuando espera que lo sea; por ejemplo, cuando ejecuta su archivo explícitamente:

```
python my_program.py
```

Sin embargo, el beneficio viene si decide `import` su archivo a otro programa (por ejemplo, si lo está escribiendo como parte de una biblioteca). Luego puede `import` su archivo, y la trampa `__main__` se asegurará de que no se ejecute ningún código inesperadamente:

```
# A new program file
import my_program        # main() is not run

# But you can run main() explicitly if you really want it to run:
my_program.main()
```

Lea Modismos en línea: https://riptutorial.com/es/python/topic/3070/modismos

Capítulo 128: Módulo aleatorio

Sintaxis

- random.seed (a = Ninguna, versión = 2) (la versión solo está disponible para Python 3.x)
- random.getstate ()
- random.setstate (estado)
- random.randint (a, b)
- random.randrange (detener)
- random.randrange (inicio, parada, paso = 1)
- elección aleatoria
- random.shuffle (x, random = random.random)
- muestra aleatoria (población, k)

Examples

Aleatorio y secuencias: barajar, selección y muestra.

```
import random
```

barajar()

Puede usar `random.shuffle()` para mezclar / aleatorizar los elementos en una secuencia **mutable e indexable** . Por ejemplo una `list` :

```
laughs = ["Hi", "Ho", "He"]

random.shuffle(laughs)     # Shuffles in-place! Don't do: laughs = random.shuffle(laughs)

print(laughs)
# Out: ["He", "Hi", "Ho"]  # Output may vary!
```

elección()

Toma un elemento aleatorio de una **secuencia** arbitraria:

```
print(random.choice(laughs))
# Out: He                  # Output may vary!
```

muestra()

Como `choice` , toma elementos aleatorios de una **secuencia** arbitraria **,** pero puedes especificar cuántos:

```
#                    |--sequence--|--number--|
print(random.sample(    laughs    ,     1    ))  # Take one element
# Out: ['Ho']                     # Output may vary!
```

no tomará el mismo elemento dos veces:

```
print(random.sample(laughs, 3))  # Take 3 random element from the sequence.
# Out: ['Ho', 'He', 'Hi']        # Output may vary!

print(random.sample(laughs, 4))  # Take 4 random element from the 3-item sequence.
```

ValueError: Muestra más grande que la población

Creación de enteros y flotadores aleatorios: randint, randrange, random y uniform

```
import random
```

randint ()

Devuelve un entero aleatorio entre `x` y `y` (inclusive):

```
random.randint(x, y)
```

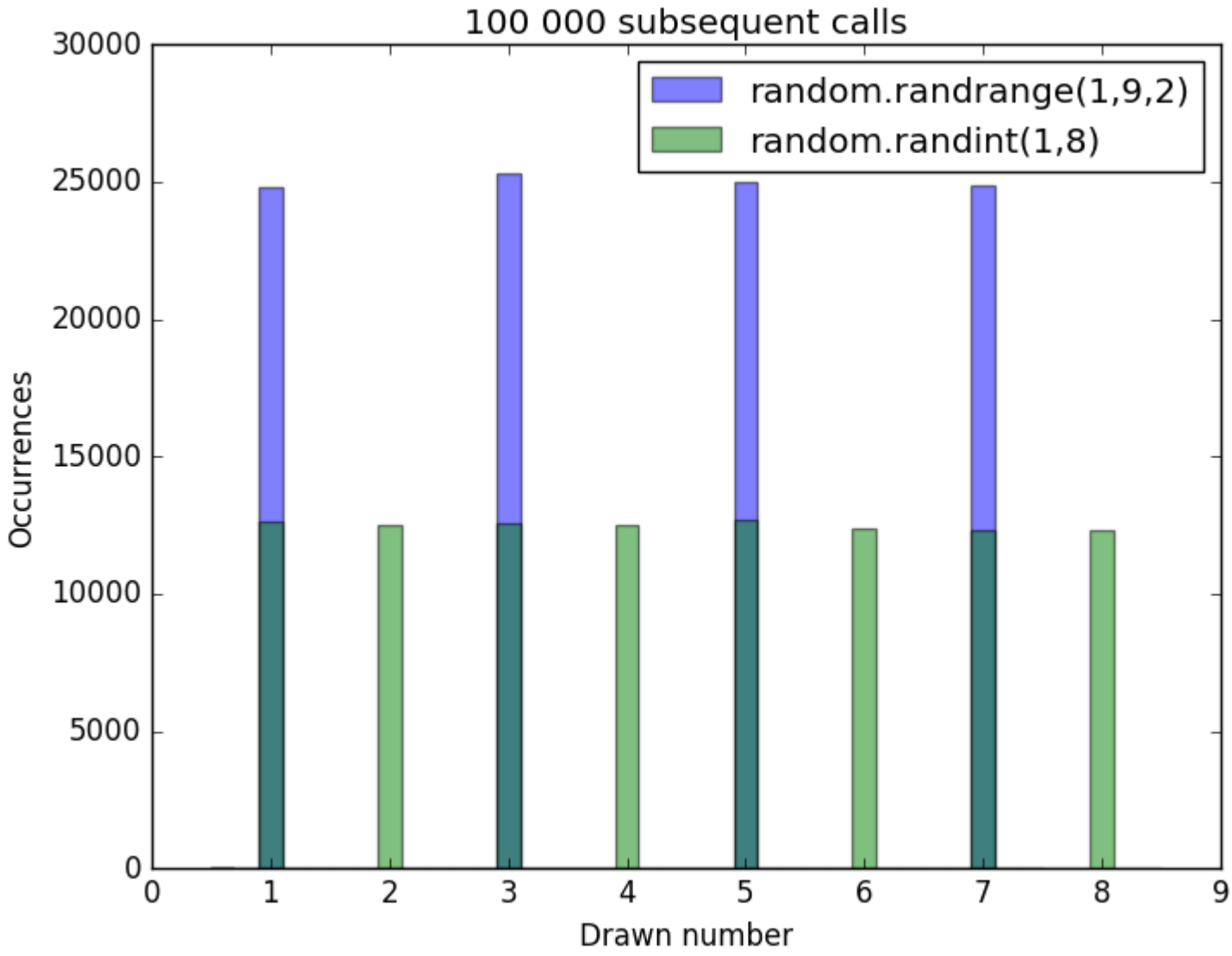
Por ejemplo obteniendo un número aleatorio entre `1` y `8` :

```
random.randint(1, 8) # Out: 8
```

randrange ()

`random.randrange` tiene la misma sintaxis que `range` y, a diferencia de `random.randint` , el último valor **no** es inclusivo:

```
random.randrange(100)       # Random integer between 0 and 99
random.randrange(20, 50)    # Random integer between 20 and 49
random.rangrange(10, 20, 3) # Random integer between 10 and 19 with step 3 (10, 13, 16 and 19)
```



aleatorio

Devuelve un número de punto flotante aleatorio entre 0 y 1:

```
random.random() # Out: 0.66486093215306317
```

uniforme

Devuelve un número de punto flotante aleatorio entre `x` y `y` (inclusive):

```
random.uniform(1, 8) # Out: 3.726062641730108
```

Números aleatorios reproducibles: semilla y estado

Establecer una semilla específica creará una serie fija de números aleatorios:

```
random.seed(5)                  # Create a fixed state
print(random.randrange(0, 10))  # Get a random integer between 0 and 9
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Restablecer la semilla creará la misma secuencia "aleatoria" de nuevo:

```
random.seed(5)                  # Reset the random module to the same fixed state.
print(random.randrange(0, 10))
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Dado que la semilla está fija, estos resultados son siempre `9` y `4` . Si no es necesario tener números específicos solo que los valores serán los mismos, también puede usar `getstate` y `setstate` para recuperar un estado anterior:

```
save_state = random.getstate()  # Get the current state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8

random.setstate(save_state)     # Reset to saved state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8
```

Para pseudoaleatorizar de nuevo la secuencia, se `seed` con `None` :

```
random.seed(None)
```

O llame al método `seed` sin argumentos:

```
random.seed()
```

Crear números aleatorios criptográficamente seguros

Por defecto, el módulo aleatorio de Python usa el `PRNG` Mersenne Twister para generar números aleatorios que, aunque son adecuados en dominios como simulaciones, no cumplen con los requisitos de seguridad en entornos más exigentes.

Para crear un número pseudoaleatorio seguro criptográficamente, se puede usar `SystemRandom` que, mediante el uso de `os.urandom` , puede actuar como un generador de número pseudoaleatorio seguro criptográficamente, CPRNG .

La forma más fácil de usarlo consiste simplemente en inicializar la clase `SystemRandom` . Los métodos proporcionados son similares a los exportados por el módulo aleatorio.

```
from random import SystemRandom
secure_rand_gen = SystemRandom()
```

Para crear una secuencia aleatoria de 10 `int` s en el rango `[0, 20]` , simplemente se puede llamar a `randrange()` :

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

Para crear un entero aleatorio en un rango dado, uno puede usar `randint` :

```
print(secure_rand_gen.randint(0, 20))
# 5
```

y, en consecuencia para todos los demás métodos. La interfaz es exactamente la misma, el único cambio es el generador de números subyacente.

También puede usar `os.urandom` directamente para obtener bytes aleatorios criptográficamente seguros.

Creando una contraseña de usuario aleatoria

Para crear una contraseña de usuario aleatoria, podemos utilizar los símbolos proporcionados en el módulo de `string` . Específicamente `punctuation` para los símbolos de puntuación, `ascii_letters` para las cartas y `digits` para dígitos:

```
from string import punctuation, ascii_letters, digits
```

Luego podemos combinar todos estos símbolos en un nombre llamado `symbols` :

```
symbols = ascii_letters + digits + punctuation
```

Elimine cualquiera de estos para crear un conjunto de símbolos con menos elementos.

Después de esto, podemos usar `random.SystemRandom` para generar una contraseña. Para una contraseña de 10 longitudes:

```
secure_random = random.SystemRandom()
password = "".join(secure_random.choice(symbols) for i in range(10))
print(password)  # '^@g;J?]M6e'
```

Tenga en cuenta que otras rutinas puestos a disposición de inmediato por el `random` módulo - como `random.choice` , `random.randint` , etc. - *no* son *adecuados* para los propósitos criptográficos.

Detrás de las cortinas, estas rutinas utilizan el PRNG de Mersenne Twister , que no satisface los requisitos de un CSPRNG . Por lo tanto, en particular, no debe usar ninguno de ellos para generar contraseñas que planea usar. Siempre use una instancia de `SystemRandom` como se muestra arriba.

Python 3.x 3.6

A partir de Python 3.6, el módulo de `secrets` está disponible, lo que expone la funcionalidad criptográficamente segura.

Al citar la documentación oficial , para generar *"una contraseña alfanumérica de diez caracteres con al menos un carácter en minúscula, al menos un carácter en mayúscula y al menos tres dígitos"* , podría:

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Decisión Binaria Aleatoria

```
import random

probability = 0.3

if random.random() < probability:
    print("Decision with probability 0.3")
else:
    print("Decision with probability 0.7")
```

Lea Módulo aleatorio en línea: https://riptutorial.com/es/python/topic/239/modulo-aleatorio

Capítulo 129: Módulo asyncio

Examples

Sintaxis de Coroutine y Delegación

Antes de que se lanzara Python `asyncio` , el módulo `asyncio` usaba generadores para imitar las llamadas asíncronas y, por lo tanto, tenía una sintaxis diferente a la versión actual de Python 3.5.

Python 3.x 3.5

Python 3.5 introdujo el `async` y `await` palabras clave. Tenga en cuenta la falta de paréntesis alrededor de la llamada `await func()` .

```
import asyncio

async def main():
    print(await func())

async def func():
    # Do time intensive stuff...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x 3.3 3.5

Antes de Python 3.5, el decorador `@asyncio.coroutine` se usaba para definir una coroutine. El rendimiento de la expresión se usó para la delegación del generador. Note los paréntesis alrededor del `yield from func()` .

```
import asyncio

@asyncio.coroutine
def main():
    print((yield from func()))

@asyncio.coroutine
def func():
    # Do time intensive stuff..
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x 3.5

Aquí hay un ejemplo que muestra cómo dos funciones pueden ejecutarse de forma asíncrona:

```
import asyncio

async def cor1():
    print("cor1 start")
    for i in range(10):
        await asyncio.sleep(1.5)
        print("cor1", i)

async def cor2():
    print("cor2 start")
    for i in range(15):
        await asyncio.sleep(1)
        print("cor2", i)

loop = asyncio.get_event_loop()
cors = asyncio.wait([cor1(), cor2()])
loop.run_until_complete(cors)
```

Ejecutores asincronos

Nota: utiliza la sintaxis de Python 3.5+ async / await

`asyncio` admite el uso de objetos `Executor` que se encuentran en `concurrent.futures` para programar tareas de forma asíncrona. Los bucles de eventos tienen la función `run_in_executor()` que toma un objeto `Executor` , un `Callable` y los parámetros del Callable.

Programando una tarea para un `Executor`

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b

async def main(loop):
    executor = ThreadPoolExecutor()
    result = await loop.run_in_executor(executor, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

Cada ciclo de eventos también tiene una ranura de `Executor` "predeterminada" que se puede asignar a un `Executor` . Para asignar un `Executor` y programar tareas desde el bucle, utilice el método `set_default_executor()` .

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b
```

```
async def main(loop):
    # NOTE: Using `None` as the first parameter designates the `default` Executor.
    result = await loop.run_in_executor(None, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.set_default_executor(ThreadPoolExecutor())
    loop.run_until_complete(main(loop))
```

Hay dos tipos principales de `Executor` en `concurrent.futures` , `ThreadPoolExecutor` y `ProcessPoolExecutor` . `ThreadPoolExecutor` contiene un conjunto de subprocesos que se pueden establecer manualmente en un número específico de subprocesos a través del constructor o por defecto el número de núcleos en los tiempos de la máquina 5. El `ThreadPoolExecutor` utiliza el conjunto de subprocesos para ejecutar tareas asignadas a él En general, es mejor en operaciones vinculadas a la CPU en lugar de operaciones vinculadas de E / S Contraste eso con el `ProcessPoolExecutor` que genera un nuevo proceso para cada tarea asignada. `ProcessPoolExecutor` solo puede tomar tareas y parámetros que son seleccionables. Las tareas no recolectables más comunes son los métodos de los objetos. Si debe programar el método de un objeto como una tarea en un `Executor` , debe usar un `ThreadPoolExecutor` .

Usando UVLoop

`uvloop` es una implementación para `asyncio.AbstractEventLoop` basada en libuv (utilizada por nodejs). Cumple con el 99% de `asyncio` funciones de `asyncio` y es mucho más rápido que el `asyncio.EventLoop` tradicional. `asyncio.EventLoop` . `uvloop` actualmente no está disponible en Windows, instálelo con `pip install uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Do your stuff here ...
```

También se puede cambiar la fábrica de bucles de eventos configurando `EventLoopPolicy` a la de `uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

Primitiva de sincronización: Evento

Concepto

Utilice un `Event` para **sincronizar la programación de múltiples coroutines** .

En pocas palabras, un evento es como el disparo en una carrera: permite que los corredores salgan de los bloques de salida.

Ejemplo

```
import asyncio

# event trigger function
def trigger(event):
    print('EVENT SET')
    event.set()  # wake up coroutines waiting

# event consumers
async def consumer_a(event):
    consumer_name = 'Consumer A'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

async def consumer_b(event):
    consumer_name = 'Consumer B'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

# event
event = asyncio.Event()

# wrap coroutines in one future
main_future = asyncio.wait([consumer_a(event),
                            consumer_b(event)])

# event loop
event_loop = asyncio.get_event_loop()
event_loop.call_later(0.1, functools.partial(trigger, event))  # trigger event in 0.1 sec

# complete main_future
done, pending = event_loop.run_until_complete(main_future)
```

Salida:

Consumidor B en espera
Consumidor A esperando
SET DE EVENTOS
Consumidor B activado
Consumidor A activado

Un simple websocket

Aquí hacemos un simple websocket eco utilizando `asyncio` . Definimos las rutinas para conectarse a un servidor y enviar / recibir mensajes. Las comunicaciones del websocket se ejecutan en una

rutina `main` , que se ejecuta mediante un bucle de eventos. Este ejemplo es modificado de una publicación anterior .

```
import asyncio
import aiohttp

session = aiohttp.ClientSession()                              # handles the context manager
class EchoWebsocket:

    async def connect(self):
        self.websocket = await session.ws_connect("wss://echo.websocket.org")

    async def send(self, message):
        self.websocket.send_str(message)

    async def receive(self):
        result = (await self.websocket.receive())
        return result.data

async def main():
    echo = EchoWebsocket()
    await echo.connect()
    await echo.send("Hello World!")
    print(await echo.receive())                                # "Hello World!"


if __name__ == '__main__':
    # The main loop
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Error común sobre asyncio

Probablemente, *la* idea errónea más común acerca de `asnycio` es que le permite ejecutar cualquier tarea en paralelo: eludir el GIL (bloqueo global del intérprete) y, por lo tanto, ejecutar trabajos de bloqueo en paralelo (en subprocesos separados). **no** lo hace!

`asyncio` (y las bibliotecas creadas para colaborar con `asyncio`) se basan en coroutines: funciones que (en colaboración) devuelven el flujo de control a la función de llamada. `asyncio.sleep` en cuenta `asyncio.sleep` en los ejemplos anteriores. este es un ejemplo de una coroutina no bloqueante que espera 'en el fondo' y devuelve el flujo de control a la función de llamada (cuando se llama con `await`). `time.sleep` es un ejemplo de una función de bloqueo. el flujo de ejecución del programa se detendrá allí y solo regresará después de que `time.sleep` haya terminado.

un ejemplo real es la biblioteca de `requests` que consiste (por el momento) solo en funciones de bloqueo. no hay concurrencia si llama a cualquiera de sus funciones dentro de `asyncio` . `aiohttp` por otro lado fue construido con `asyncio` en mente. sus coroutines correrán concurrentemente.

- Si tiene tareas vinculadas a la CPU de larga ejecución que le gustaría ejecutar en paralelo, `asyncio` **no** es para usted. Para eso necesitas `threads` o `multiprocessing` .
- Si tiene trabajos en ejecución enlazados a IO, *puede* ejecutarlos simultáneamente usando `asyncio` .

Lea Módulo asyncio en línea: https://riptutorial.com/es/python/topic/1319/modulo-asyncio

Capítulo 130: Módulo de cola

Introducción

El módulo Queue implementa colas multi-productor, multi-consumidor. Es especialmente útil en la programación de subprocesos cuando la información se debe intercambiar de forma segura entre varios subprocesos. Hay tres tipos de colas que proporciona el módulo de colas, que son las siguientes: 1. Cola 2. LifoQueue 3. Excepción de PriorityQueue que podría venir: 1. Completa (desbordamiento de cola) 2. Vacía (desbordamiento de cola)

Examples

Ejemplo simple

```
from Queue import Queue

question_queue = Queue()

for x in range(1,10):
    temp_dict = ('key', x)
    question_queue.put(temp_dict)

while(not question_queue.empty()):
    item = question_queue.get()
    print(str(item))
```

Salida:

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

Lea Módulo de cola en línea: https://riptutorial.com/es/python/topic/8339/modulo-de-cola

Capítulo 131: Módulo de colecciones

Introducción

El paquete de `collections` incorporado proporciona varios tipos de colección flexibles y especializados que tienen un alto rendimiento y ofrecen alternativas a los tipos de colección generales de `dict` , `list` , `tuple` y `set` . El módulo también define clases básicas abstractas que describen diferentes tipos de funcionalidad de colección (como `MutableSet` y `ItemsView`).

Observaciones

Hay otros tres tipos disponibles en el módulo de **colecciones** , a saber:

1. UserDict
2. Lista de usuarios
3. UserString

Cada una de ellas actúa como una envoltura alrededor del objeto atado, por ejemplo, *UserDict* actúa como una envoltura alrededor de un objeto *dict* . En cada caso, la clase simula su tipo nombrado. El contenido de la instancia se mantiene en un objeto de tipo regular, al que se puede acceder a través del atributo de datos de la instancia de contenedor. En cada uno de estos tres casos, la necesidad de estos tipos ha sido parcialmente suplantada por la capacidad de subclasificar directamente del tipo básico; sin embargo, puede ser más fácil trabajar con la clase contenedora porque el tipo subyacente es accesible como un atributo.

Examples

colecciones.

Contador es una subclase de dict que le permite contar objetos fácilmente. Tiene métodos de utilidad para trabajar con las frecuencias de los objetos que está contando.

```
import collections
counts = collections.Counter([1,2,3])
```

el código anterior crea un objeto, cuenta, que tiene las frecuencias de todos los elementos pasados al constructor. Este ejemplo tiene el valor `Counter({1: 1, 2: 1, 3: 1})`

Ejemplos de constructor

Contador de cartas

```
>>> collections.Counter('Happy Birthday')
Counter({'a': 2, 'p': 2, 'y': 2, 'i': 1, 'r': 1, 'B': 1, ' ': 1, 'H': 1, 'd': 1, 'h': 1, 't':
1})
```

Contador de palabras

```
>>> collections.Counter('I am Sam Sam I am That Sam-I-am That Sam-I-am! I do not like that
Sam-I-am'.split())
Counter({'I': 3, 'Sam': 2, 'Sam-I-am': 2, 'That': 2, 'am': 2, 'do': 1, 'Sam-I-am!': 1, 'that':
1, 'not': 1, 'like': 1})
```

Recetas

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

Consigue la cuenta del elemento individual.

```
>>> c['a']
4
```

Establecer la cuenta del elemento individual

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

Obtener el número total de elementos en el contador (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues())  # negative numbers are counted!
3
```

Obtener elementos (solo se mantienen los que tienen un contador positivo)

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Eliminar claves con 0 o valor negativo.

```
>>> c - collections.Counter()
Counter({'a': 4, 'b': 2})
```

Quitar todo

```
>>> c.clear()
>>> c
Counter()
```

Añadir eliminar elementos individuales

```
>>> c.update({'a': 3, 'b':3})
>>> c.update({'a': 2, 'c':2})  # adds to existing, sets if they don't exist
>>> c
Counter({'a': 5, 'b': 3, 'c': 2})
>>> c.subtract({'a': 3, 'b': 3, 'c': 3})  # subtracts (negative values are allowed)
```

```
>>> c
Counter({'a': 2, 'b': 0, 'c': -1})
```

colecciones.defaultdict

collections.defaultdict (default_factory) devuelve una subclase de `dict` que tiene un valor predeterminado para las claves que faltan. El argumento debe ser una función que devuelve el valor predeterminado cuando se llama sin argumentos. Si no se pasa nada, el valor predeterminado es `None` .

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

devuelve una referencia a un defaultdict que creará un objeto de cadena con su método default_factory.

Un uso típico de `defaultdict` es usar uno de los tipos incorporados como `str` , `int` , `list` o `dict` como default_factory, ya que estos devuelven tipos vacíos cuando se les llama sin argumentos:

```
>>> str()
''
>>> int()
0
>>> list
[]
```

Llamar al valor predeterminado con una clave que no existe no produce un error como lo haría en un diccionario normal.

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

Otro ejemplo con `int` :

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2  # No errors should occur
>>> fruit_counts
default_dict(int, {'apple': 2})
>>> fruit_counts['banana']  # No errors should occur
0
>>> fruit_counts  # A new key is created
default_dict(int, {'apple': 2, 'banana': 0})
```

Los métodos de diccionario normales funcionan con el diccionario predeterminado

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

El uso de `list` como default_factory creará una lista para cada nueva clave.

```
>>> s = [('NC', 'Raleigh'), ('VA', 'Richmond'), ('WA', 'Seattle'), ('NC', 'Asheville')]
>>> dd = collections.defaultdict(list)
>>> for k, v in s:
...     dd[k].append(v)
>>> dd
defaultdict(<class 'list'>,
    {'VA': ['Richmond'],
     'NC': ['Raleigh', 'Asheville'],
     'WA': ['Seattle']})
```

colecciones.OrdenedDict

El orden de las claves en los diccionarios de Python es arbitrario: no se rigen por el orden en el que se agregan.

Por ejemplo:

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
```

(El orden arbitrario implícito arriba significa que puede obtener resultados diferentes con el código anterior al que se muestra aquí).

El orden en que aparecen las teclas es el orden en el que se iterarían, por ejemplo, utilizando un bucle `for` .

La clase `collections.OrderedDict` proporciona objetos de diccionario que conservan el orden de las claves. `OrderedDict` s se puede crear como se muestra a continuación con una serie de artículos ordenados (aquí, una lista de pares clave-valor de tupla):

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('foo', 5), ('bar', 6)])
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6)])
>>> d['baz'] = 7
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7)])
>>> d['foobar'] = 8
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

O podemos crear un `OrderedDict` vacío y luego agregar elementos:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

La `OrderedDict` través de un `OrderedDict` permite el acceso de claves en el orden en que se agregaron.

¿Qué sucede si asignamos un nuevo valor a una clave existente?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

La clave conserva su lugar original en el `OrderedDict` .

colecciones.namedu tupla

Defina un nuevo tipo de `Person` usando `namedtuple` como este:

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

El segundo argumento es la lista de atributos que tendrá la tupla. También puede enumerar estos atributos como espacios o cadenas separadas por comas:

```
Person = namedtuple('Person', 'age, height, name')
```

o

```
Person = namedtuple('Person', 'age height name')
```

Una vez definido, se puede crear una instancia de una tupla con nombre llamando al objeto con los parámetros necesarios, por ejemplo:

```
dave = Person(30, 178, 'Dave')
```

Los argumentos con nombre también se pueden utilizar:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Ahora puedes acceder a los atributos de la pareja nombrada:

```
print(jack.age)  # 30
print(jack.name)  # 'Jack S.'
```

El primer argumento para el constructor de elementos nombrados (en nuestro ejemplo `'Person'`) es el `typename` . Es típico usar la misma palabra para el constructor y el nombre tipográfico, pero

pueden ser diferentes:

```
Human = namedtuple('Person',  'age, height, name')
dave = Human(30, 178, 'Dave')
print(dave)  # yields: Person(age=30, height=178, name='Dave')
```

colecciones.deque

Devuelve un nuevo objeto `deque` inicializado de izquierda a derecha (utilizando append ()) con datos de iterable. Si iterable no está especificado, el nuevo `deque` está vacío.

Deques es una generalización de pilas y colas (el nombre se pronuncia "deck" y es la abreviatura de "cola doble"). Deques es compatible con subprocesos seguros y eficientes con la memoria y hace estallar desde cualquier lado del `deque` con aproximadamente el mismo rendimiento O (1) en cualquier dirección.

Aunque los objetos de la lista admiten operaciones similares, están optimizados para operaciones rápidas de longitud fija e incurren en costos de movimiento de memoria O (n) para operaciones pop (0) e insertar (0, v) que cambian el tamaño y la posición de la representación de datos subyacente .

Nuevo en la versión 2.4.

Si `maxlen` no está especificado o es `None` , los deques pueden crecer hasta una longitud arbitraria. De lo contrario, el `deque` se limita a la longitud máxima especificada. Una vez que el `deque` longitud acotada está lleno, cuando se agregan nuevos elementos, un número correspondiente de elementos se descarta del extremo opuesto. Los deques de longitud limitada proporcionan una funcionalidad similar al filtro de cola en Unix. También son útiles para rastrear transacciones y otros grupos de datos donde solo interesa la actividad más reciente.

Cambiado en la versión 2.6: Se agregó el parámetro maxlen.

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
```

```
>>> d[-1]                           # peek at rightmost item
'i'

>>> list(reversed(d))               # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                        # search the deque
True
>>> d.extend('jkl')                 # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                     # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

Fuente: https://docs.python.org/2/library/collections.html

colecciones.ChainMap

`ChainMap` es nuevo en la **versión 3.3**

Devuelve un nuevo objeto `ChainMap` dado un número de `maps` . Este objeto agrupa varios dicts u otras asignaciones para crear una vista única y actualizable.

`ChainMap` son útiles para administrar contextos y superposiciones anidadas. Un ejemplo en el mundo de python se encuentra en la implementación de la clase de `Context` en el motor de plantillas de Django. Es útil para vincular rápidamente varias asignaciones para que el resultado se pueda tratar como una sola unidad. A menudo es mucho más rápido que crear un nuevo diccionario y ejecutar varias llamadas de `update()` .

Cada vez que uno tiene una cadena de valores de búsqueda, puede haber un caso para `ChainMap` . Un ejemplo incluye tener valores especificados por el usuario y un diccionario de valores predeterminados. Otro ejemplo son los mapas de parámetros `POST` y `GET` que se encuentran en el uso web, por ejemplo, Django o Flask. Mediante el uso de `ChainMap` uno devuelve una vista combinada de dos diccionarios distintos.

La lista de parámetros de `maps` se ordena desde la primera búsqueda hasta la última búsqueda. Las búsquedas buscan sucesivamente las asignaciones subyacentes hasta que se encuentra una clave. Por el contrario, las escrituras, actualizaciones y eliminaciones solo funcionan en la primera

asignación.

```
import collections

# define two dictionaries with at least some keys overlapping.
dict1 = {'apple': 1, 'banana': 2}
dict2 = {'coconut': 1, 'date': 1, 'apple': 3}

# create two ChainMaps with different ordering of those dicts.
combined_dict = collections.ChainMap(dict1, dict2)
reverse_ordered_dict = collections.ChainMap(dict2, dict1)
```

Tenga en cuenta el impacto del orden en el que el valor se encuentra primero en la búsqueda posterior

```
for k, v in combined_dict.items():
    print(k, v)

date 1
apple 1
banana 2
coconut 1

for k, v in reverse_ordered_dict.items():
    print(k, v)

date 1
apple 3
banana 2
coconut 1
```

Lea Módulo de colecciones en línea: https://riptutorial.com/es/python/topic/498/modulo-de-colecciones

Capítulo 132: Módulo de funciones

Examples

parcial

La función `partial` crea una aplicación de función parcial desde otra función. Se utiliza para *vincular* valores a algunos de los argumentos de la función (o argumentos de palabras clave) y producir una *llamada* sin los argumentos ya definidos.

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('ca11ab1e')
3390155550
```

`partial()` , como su nombre indica, permite una evaluación parcial de una función. Veamos el siguiente ejemplo:

```
In [2]: from functools import partial

In [3]: def f(a, b, c, x):
   ...:     return 1000*a + 100*b + 10*c + x
   ...:

In [4]: g = partial(f, 1, 1, 1)

In [5]: print g(2)
1112
```

Cuando se crea `g` , `f` , que toma cuatro argumentos (`a, b, c, x`), también se evalúa parcialmente para los primeros tres argumentos, `a, b, c,` Evaluación de `f` se completa cuando `g` es llamado, `g(2)` , que pasa el cuarto argumento a `f` .

Una forma de pensar en `partial` es un registro de desplazamiento; empujando en un argumento en el momento en alguna función. `partial` es útil para los casos en que los datos entran como flujo y no podemos pasar más de un argumento.

ordenamiento total

Cuando queremos crear una clase ordenable, normalmente necesitamos definir los métodos `__eq()__` , `__lt__()` , `__le__()` , `__gt__()` y `__ge__()` .

El decorador de `total_ordering` , aplicado a una clase, permite la definición de `__eq__()` y solo uno entre `__lt__()` , `__le__()` , `__gt__()` y `__ge__()` , y aún permite todas las operaciones de ordenamiento en la clase.

```
@total_ordering
```

```
class Employee:

    ...

    def __eq__(self, other):
        return ((self.surname, self.name) == (other.surname, other.name))

    def __lt__(self, other):
        return ((self.surname, self.name) < (other.surname, other.name))
```

El decorador utiliza una composición de los métodos proporcionados y las operaciones algebraicas para derivar los otros métodos de comparación. Por ejemplo, si definimos `__lt__()` y `__eq()__` y queremos derivar `__gt__()` , simplemente podemos verificar `not __lt__() and not __eq()__` .

Nota : la función `total_ordering` solo está disponible desde Python 2.7.

reducir

En Python 3.x, la función de `reduce` ya explicada aquí se ha eliminado de las funciones integradas y ahora debe importarse desde `functools` .

```
from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))
```

lru_cache

El decorador `@lru_cache` se puede usar para envolver una función costosa y de uso intensivo de computación con un caché de uso menos reciente . Esto permite que las llamadas de función se memoricen, de modo que las llamadas futuras con los mismos parámetros puedan retornar instantáneamente en lugar de tener que volver a calcularse.

```
@lru_cache(maxsize=None)  # Boundless cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)
```

En el ejemplo anterior, el valor de `fibonacci(3)` solo se calcula una vez, mientras que si `fibonacci` no tuviera un caché LRU, `fibonacci(3)` se habría calculado más de 230 veces. Por lo tanto, `@lru_cache` es especialmente bueno para funciones recursivas o programación dinámica, donde una función costosa podría llamarse varias veces con los mismos parámetros exactos.

`@lru_cache` tiene dos argumentos

- `maxsize` : Número de llamadas a guardar. Cuando el número de llamadas únicas supera el `maxsize` , el caché LRU eliminará las llamadas menos utilizadas recientemente.
- `typed` (agregado en 3.3): marca para determinar si los argumentos equivalentes de

diferentes tipos pertenecen a diferentes registros de caché (es decir, si `3.0` y `3` cuentan como argumentos diferentes)

Podemos ver las estadísticas de caché también:

```
>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

NOTA : Ya que `@lru_cache` usa los diccionarios para almacenar en caché los resultados, todos los parámetros para la función deben estar habilitados para que la caché funcione.

Documentos oficiales de Python para `@lru_cache` . `@lru_cache` fue agregado en 3.2.

cmp_to_key

Python cambió sus métodos de clasificación para aceptar una función clave. Esas funciones toman un valor y devuelven una clave que se usa para ordenar las matrices.

Las antiguas funciones de comparación utilizadas para tomar dos valores y devolver -1, 0 o +1 si el primer argumento es pequeño, igual o mayor que el segundo argumento, respectivamente. Esto es incompatible con la nueva función clave.

Ahí es donde entra `functools.cmp_to_key` :

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

Ejemplo tomado y adaptado de la documentación de la biblioteca estándar de Python .

Lea Módulo de funciones en línea: https://riptutorial.com/es/python/topic/2492/modulo-de-funciones

Capítulo 133: Módulo de matemáticas

Examples

Redondeo: redondo, suelo, ceil, trunc

Además de la función `round` incorporada, el módulo `math` proporciona las funciones `floor` , `ceil` y `trunc` .

```
x = 1.55
y = -1.55

# round to the nearest integer
round(x)       #  2
round(y)       # -2

# the second argument gives how many decimal places to round to (defaults to 0)
round(x, 1)    #  1.6
round(y, 1)    # -1.6

# math is a module so import it first, then use it.
import math

# get the largest integer less than x
math.floor(x)  #  1
math.floor(y)  # -2

# get the smallest integer greater than x
math.ceil(x)   #  2
math.ceil(y)   # -1

# drop fractional part of x
math.trunc(x)  #  1, equivalent to math.floor for positive numbers
math.trunc(y)  # -1, equivalent to math.ceil for negative numbers
```

Python 2.x 2.7

`floor` , el `ceil` , el `trunc` y el `round` siempre devuelven un `float` .

```
round(1.3)  # 1.0
```

`round` siempre rompe empates lejos de cero.

```
round(0.5)  # 1.0
round(1.5)  # 2.0
```

Python 3.x 3.0

`floor` , `ceil` y `trunc` siempre devuelven un valor `Integral` , mientras que `round` devuelve un valor `Integral` si se llama con un argumento.

```
round(1.3)      # 1
round(1.33, 1)  # 1.3
```

Rupturas `round` empates hacia el número par más cercano. Esto corrige el sesgo hacia números más grandes cuando se realizan una gran cantidad de cálculos.

```
round(0.5)  # 0
round(1.5)  # 2
```

¡Advertencia!

Al igual que con cualquier representación de punto flotante, algunas fracciones *no se pueden representar exactamente* . Esto puede llevar a algún comportamiento de redondeo inesperado.

```
round(2.675, 2)  # 2.67, not 2.68!
```

Advertencia sobre la división de números negativos en el piso, corte y número entero

Python (y C ++ y Java) se alejan de cero para los números negativos. Considerar:

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

Logaritmos

`math.log(x)` da el logaritmo natural (base `e`) de `x` .

```
math.log(math.e)  # 1.0
math.log(1)       # 0.0
math.log(100)     # 4.605170185988092
```

`math.log` puede perder precisión con números cercanos a 1, debido a las limitaciones de los números de punto flotante. Para calcular con precisión los registros cercanos a 1, use `math.log1p` , que evalúa el logaritmo natural de 1 más el argumento:

```
math.log(1 + 1e-20)  # 0.0
math.log1p(1e-20)    # 1e-20
```

`math.log10` puede usarse para logs base 10:

```
math.log10(10)  # 1.0
```

Python 2.x 2.3.0

Cuando se usa con dos argumentos, `math.log(x, base)` da el logaritmo de `x` en la `base` dada (es

decir, `log(x) / log(base)` .

```
math.log(100, 10) # 2.0
math.log(27, 3)   # 3.0
math.log(1, 10)   # 0.0
```

Copiando carteles

En Python 2.6 y superior, `math.copysign(x, y)` devuelve `x` con el signo de `y` . El valor devuelto es siempre un `float` .

Python 2.x 2.6

```
math.copysign(-2, 3)    # 2.0
math.copysign(3, -3)    # -3.0
math.copysign(4, 14.2)  # 4.0
math.copysign(1, -0.0)  # -1.0, on a platform which supports signed zero
```

Trigonometría

Cálculo de la longitud de la hipotenusa.

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)
# Out: 4.47213595499958
```

Convertir grados a / desde radianes

Todas las funciones `math` esperan **radianes,** por lo que necesitas convertir grados a radianes:

```
math.radians(45)              # Convert 45 degrees to radians
# Out: 0.7853981633974483
```

Todos los resultados de las funciones trigonométricas inversas devuelven el resultado en radianes, por lo que es posible que deba volver a convertirlo en grados:

```
math.degrees(math.asin(1))    # Convert the result of asin to degrees
# Out: 90.0
```

Funciones seno, coseno, tangente e inversa.

```
# Sine and arc sine
math.sin(math.pi / 2)
# Out: 1.0
math.sin(math.radians(90))   # Sine of 90 degrees
# Out: 1.0

math.asin(1)
# Out: 1.5707963267948966    # "= pi / 2"
math.asin(1) / math.pi
# Out: 0.5
```

```
# Cosine and arc cosine:
math.cos(math.pi / 2)
# Out: 6.123233995736766e-17
# Almost zero but not exactly because "pi" is a float with limited precision!

math.acos(1)
# Out: 0.0

# Tangent and arc tangent:
math.tan(math.pi/2)
# Out: 1.633123935319537e+16
# Very large but not exactly "Inf" because "pi" is a float with limited precision
```

Python 3.x 3.5

```
math.atan(math.inf)
# Out: 1.5707963267948966 # This is just "pi / 2"
```

```
math.atan(float('inf'))
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Aparte de `math.atan` también hay una función `math.atan2` dos argumentos, que calcula el cuadrante correcto y evita los escollos de la división por cero:

```
math.atan2(1, 2)   # Equivalent to "math.atan(1/2)"
# Out: 0.4636476090008061 # ≈ 26.57 degrees, 1st quadrant

math.atan2(-1, -2) # Not equal to "math.atan(-1/-2)" == "math.atan(1/2)"
# Out: -2.677945044588987 # ≈ -153.43 degrees (or 206.57 degrees), 3rd quadrant

math.atan2(1, 0)   # math.atan(1/0) would raise ZeroDivisionError
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Seno hiperbólico, coseno y tangente.

```
# Hyperbolic sine function
math.sinh(math.pi) # = 11.548739357257746
math.asinh(1)      # = 0.8813735870195429

# Hyperbolic cosine function
math.cosh(math.pi) # = 11.591953275521519
math.acosh(1)      # = 0.0

# Hyperbolic tangent function
math.tanh(math.pi) # = 0.99627207622075
math.atanh(0.5)    # = 0.5493061443340549
```

Constantes

`math` módulos `math` incluyen dos constantes matemáticas de uso común.

- `math.pi` - La constante matemática pi
- `math.e` - La constante matemática *e* (base del logaritmo natural)

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>>
```

Python 3.5 y superior tienen constantes para infinito y NaN ("no es un número"). La sintaxis anterior de pasar una cadena a `float()` aún funciona.

Python 3.x 3.5

```
math.inf == float('inf')
# Out: True

-math.inf == float('-inf')
# Out: True

# NaN never compares equal to anything, even itself
math.nan == float('nan')
# Out: False
```

Números imaginarios

Los números imaginarios en Python están representados por una "j" o "J" detrás del número objetivo.

```
1j         # Equivalent to the square root of -1.
1j * 1j    # = (-1+0j)
```

Infinito y NaN ("no es un número")

En todas las versiones de Python, podemos representar infinito y NaN ("no un número") de la siguiente manera:

```
pos_inf = float('inf')     # positive infinity
neg_inf = float('-inf')    # negative infinity
not_a_num = float('nan')   # NaN ("not a number")
```

En Python 3.5 y superior, también podemos usar las constantes definidas `math.inf` y `math.nan` :

Python 3.x 3.5

```
pos_inf = math.inf
neg_inf = -math.inf
not_a_num = math.nan
```

Las representaciones de cadena se muestran como `inf` y `-inf` y `nan` :

```
pos_inf, neg_inf, not_a_num
# Out: (inf, -inf, nan)
```

Podemos probar el infinito positivo o negativo con el método `isinf` :

```
math.isinf(pos_inf)
# Out: True

math.isinf(neg_inf)
# Out: True
```

Podemos probar específicamente el infinito positivo o el infinito negativo por comparación directa:

```
pos_inf == float('inf')    # or  == math.inf in Python 3.5+
# Out: True

neg_inf == float('-inf')   # or  == -math.inf in Python 3.5+
# Out: True

neg_inf == pos_inf
# Out: False
```

Python 3.2 y superior también permite verificar la finitud:

Python 3.x 3.2

```
math.isfinite(pos_inf)
# Out: False

math.isfinite(0.0)
# Out: True
```

Los operadores de comparación funcionan como se espera para el infinito positivo y negativo:

```
import sys

sys.float_info.max
# Out: 1.7976931348623157e+308  (this is system-dependent)

pos_inf > sys.float_info.max
# Out: True

neg_inf < -sys.float_info.max
# Out: True
```

Pero si una expresión aritmética produce un valor más grande que el máximo que se puede representar como un `float` , se convertirá en infinito:

```
pos_inf == sys.float_info.max * 1.0000001
# Out: True

neg_inf == -sys.float_info.max * 1.0000001
# Out: True
```

Sin embargo, la división por cero no da un resultado de infinito (o infinito negativo cuando sea apropiado), sino que genera una excepción `ZeroDivisionError` .

```
try:
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Division by zero")

# Out: Division by zero
```

Las operaciones aritméticas en el infinito solo dan resultados infinitos, o algunas veces NaN:

```
-5.0 * pos_inf == neg_inf
# Out: True

-5.0 * neg_inf == pos_inf
# Out: True

pos_inf * neg_inf == neg_inf
# Out: True

0.0 * pos_inf
# Out: nan

0.0 * neg_inf
# Out: nan

pos_inf / pos_inf
# Out: nan
```

NaN nunca es igual a nada, ni siquiera a sí mismo. Podemos probar que es con el método `isnan` :

```
not_a_num == not_a_num
# Out: False

math.isnan(not_a_num)
Out: True
```

NaN siempre se compara como "no igual", pero nunca menor o mayor que:

```
not_a_num != 5.0   # or any random value
# Out: True

not_a_num > 5.0   or   not_a_num < 5.0   or   not_a_num == 5.0
# Out: False
```

Las operaciones aritméticas en NaN siempre dan NaN. Esto incluye la multiplicación por -1: no hay "NaN negativo".

```
5.0 * not_a_num
# Out: nan

float('-nan')
# Out: nan
```

Python 3.x 3.5

```
-math.nan
# Out: nan
```

Hay una diferencia sutil entre las versiones `float` antiguas de NaN e infinito y las constantes de la biblioteca `math` Python 3.5+:

Python 3.x 3.5

```
math.inf is math.inf, math.nan is math.nan
# Out: (True, True)

float('inf') is float('inf'), float('nan') is float('nan')
# Out: (False, False)
```

Pow para una exponenciación más rápida

Usando el módulo timeit desde la línea de comando:

```
> python -m timeit 'for x in xrange(50000): b = x**3'
10 loops, best of 3: 51.2 msec per loop
> python -m timeit 'from math import pow' 'for x in xrange(50000): b = pow(x,3)'
100 loops, best of 3: 9.15 msec per loop
```

El operador `**` integrado a menudo es útil, pero si el rendimiento es esencial, use math.pow. Sin embargo, asegúrese de tener en cuenta que pow devuelve flotantes, incluso si los argumentos son enteros:

```
> from math import pow
> pow(5,5)
3125.0
```

Números complejos y el módulo cmath.

El módulo `cmath` es similar al módulo `math` , pero define funciones adecuadamente para el plano complejo.

En primer lugar, los números complejos son un tipo numérico que forma parte del lenguaje Python en lugar de ser proporcionado por una clase de biblioteca. Por lo tanto, no necesitamos `import cmath` para expresiones aritméticas ordinarias.

Tenga en cuenta que usamos `j` (o `J`) y no `i` .

```
z = 1 + 3j
```

Debemos usar `1j` ya que `j` sería el nombre de una variable en lugar de un literal numérico.

```
1j * 1j
Out: (-1+0j)

1j ** 1j
```

```
# Out: (0.20787957635076193+0j)     # "i to the i"  ==  math.e ** -(math.pi/2)
```

Tenemos la parte `real` y la parte `imag` (imaginaria), así como el complejo `conjugate` :

```
# real part and imaginary part are both float type
z.real, z.imag
# Out: (1.0, 3.0)

z.conjugate()
# Out: (1-3j)    # z.conjugate() == z.real - z.imag * 1j
```

Las funciones integradas `abs` y `complex` también son parte del lenguaje en sí y no requieren ninguna importación:

```
abs(1 + 1j)
# Out: 1.4142135623730951     # square root of 2

complex(1)
# Out: (1+0j)

complex(imag=1)
# Out: (1j)

complex(1, 1)
# Out: (1+1j)
```

La función `complex` puede tomar una cadena, pero no puede tener espacios:

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
# Exception: ValueError: complex() arg is a malformed string
```

Pero para la mayoría de las funciones necesitamos el módulo, por ejemplo, `sqrt` :

```
import cmath

cmath.sqrt(-1)
# Out: 1j
```

Naturalmente, el comportamiento de `sqrt` es diferente para números complejos y números reales. En `math` no complejas `math` la raíz cuadrada de un número negativo genera una excepción:

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

Se proporcionan funciones para convertir hacia y desde coordenadas polares:

```
cmath.polar(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # == (sqrt(1 + 1), atan2(1, 1))
```

```
abs(1 + 1j), cmath.phase(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # same as previous calculation

cmath.rect(math.sqrt(2), math.atan(1))
# Out: (1.0000000000000002+1.0000000000000002j)
```

El campo matemático del análisis complejo está más allá del alcance de este ejemplo, pero muchas funciones en el plano complejo tienen un "corte de rama", generalmente a lo largo del eje real o el eje imaginario. La mayoría de las plataformas modernas admiten el "cero firmado" como se especifica en IEEE 754, que proporciona continuidad de esas funciones en ambos lados del corte de rama. El siguiente ejemplo es de la documentación de Python:

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
# Out: -3.141592653589793
```

El módulo `cmath` también proporciona muchas funciones con contrapartes directas del módulo `math` .

Además de `sqrt` , existen versiones complejas de `exp` , `log` , `log10` , las funciones trigonométricas y sus inverses (`sin` , `cos` , `tan` , `asin` , `acos` , `atan`), y las funciones hiperbólicas y sus inverses (`sinh` , `cosh` , `tanh` , `asinh` , `acosh` , `atanh`). Sin embargo, tenga en cuenta que no hay una contraparte compleja de `math.atan2` , la forma de arctangente de dos argumentos.

```
cmath.log(1+1j)
# Out: (0.3465735902799726+0.7853981633974483j)

cmath.exp(1j * cmath.pi)
# Out: (-1+1.2246467991473532e-16j)   # e to the i pi == -1, within rounding error
```

Se proporcionan las constantes `pi` y `e` . Tenga en cuenta que estos son `float` y no `complex` .

```
type(cmath.pi)
# Out: <class 'float'>
```

El módulo `cmath` también proporciona versiones complejas de `isinf` , y (para Python 3.2+) es `isfinite` . Ver " Infinito y NaN ". Un número complejo se considera infinito si su parte real o su parte imaginaria es infinita.

```
cmath.isinf(complex(float('inf'), 0.0))
# Out: True
```

Asimismo, el módulo `cmath` proporciona una versión compleja de `isnan` . Ver " Infinito y NaN ". Un número complejo se considera "no un número" si su parte real o su parte imaginaria no es "un número".

```
cmath.isnan(0.0, float('nan'))
```

```
# Out: True
```

Tenga en cuenta que no hay `cmath` contraparte de `math.nan` constantes `math.inf` y `math.nan` (de Python 3.5 y superior)

Python 3.x 3.5

```
cmath.isinf(complex(0.0, math.inf))
# Out: True

cmath.isnan(complex(math.nan, 0.0))
# Out: True

cmath.inf
# Exception: AttributeError: module 'cmath' has no attribute 'inf'
```

En Python 3.5 y superior, hay una `isclose` método en ambos `cmath` y `math` módulos.

Python 3.x 3.5

```
z = cmath.rect(*cmath.polar(1+1j))

z
# Out: (1.0000000000000002+1.0000000000000002j)

cmath.isclose(z, 1+1j)
# True
```

Lea Módulo de matemáticas en línea: https://riptutorial.com/es/python/topic/230/modulo-de-matematicas

Capítulo 134: Módulo de navegador web

Introducción

De acuerdo con la documentación estándar de Python, el módulo del navegador web proporciona una interfaz de alto nivel para permitir la visualización de documentos basados en la Web a los usuarios. Este tema explica y demuestra el uso adecuado del módulo de navegador web.

Sintaxis

- `webbrowser.open(url, new=0, autoraise=False)`
- `webbrowser.open_new(url)`
- `webbrowser.open_new_tab(url)`
- `webbrowser.get(usage=None)`
- `webbrowser.register(name, constructor, instance=None)`

Parámetros

Parámetro	Detalles
`webbrowser.open()`	
url	La URL para abrir en el navegador web.
nuevo	0 abre la URL en la pestaña existente, 1 abre en una nueva ventana, 2 abre en nueva pestaña
autoraise	si se establece en Verdadero, la ventana se moverá sobre las otras ventanas
`webbrowser.open_new()`	
url	La URL para abrir en el navegador web.
`webbrowser.open_new_tab()`	
url	La URL para abrir en el navegador web.
`webbrowser.get()`	
utilizando	el navegador para usar
`webbrowser.register()`	
url	nombre del navegador
constructor	ruta al navegador ejecutable (ayuda)
ejemplo	Una instancia de un navegador web devuelto por el método

Parámetro	Detalles
	`webbrowser.get()`

Observaciones

La siguiente tabla enumera los tipos de navegador predefinidos. La columna de la izquierda son nombres que se pueden pasar al método `webbrowser.get()` y la columna de la derecha enumera los nombres de clase para cada tipo de navegador.

Escribe un nombre	Nombre de la clase
`'mozilla'`	`Mozilla('mozilla')`
`'firefox'`	`Mozilla('mozilla')`
`'netscape'`	`Mozilla('netscape')`
`'galeon'`	`Galeon('galeon')`
`'epiphany'`	`Galeon('epiphany')`
`'skipstone'`	`BackgroundBrowser('skipstone')`
`'kfmclient'`	`Konqueror()`
`'konqueror'`	`Konqueror()`
`'kfm'`	`Konqueror()`
`'mosaic'`	`BackgroundBrowser('mosaic')`
`'opera'`	`Opera()`
`'grail'`	`Grail()`
`'links'`	`GenericBrowser('links')`
`'elinks'`	`Elinks('elinks')`
`'lynx'`	`GenericBrowser('lynx')`
`'w3m'`	`GenericBrowser('w3m')`
`'windows-default'`	`WindowsDefault`
`'macosx'`	`MacOSX('default')`
`'safari'`	`MacOSX('safari')`
`'google-chrome'`	`Chrome('google-chrome')`
`'chrome'`	`Chrome('chrome')`
`'chromium'`	`Chromium('chromium')`
`'chromium-browser'`	`Chromium('chromium-browser')`

Examples

Abrir una URL con el navegador predeterminado

Para abrir simplemente una URL, use el método `webbrowser.open()` :

```
import webbrowser
webbrowser.open("http://stackoverflow.com")
```

Si una ventana del navegador está actualmente abierta, el método abrirá una nueva pestaña en la URL especificada. Si no hay ninguna ventana abierta, el método abrirá el navegador predeterminado del sistema operativo y navegará a la URL en el parámetro. El método abierto soporta los siguientes parámetros:

- `url` : la URL que se abrirá en el navegador web (cadena) **[requerido]**
- `new` : 0 se abre en la pestaña existente, 1 abre una nueva ventana, 2 abre una nueva pestaña (entero) **[predeterminado 0]**
- `autoraise` : si se establece en Verdadero, la ventana se moverá sobre las ventanas de otras aplicaciones (Booleano) **[predeterminado Falso]**

Tenga en cuenta que los argumentos `new` y `autoraise` rara vez funcionan, ya que la mayoría de los navegadores modernos rechazan estos comandos.

El navegador web también puede intentar abrir las URL en nuevas ventanas con el método `open_new` :

```
import webbrowser
webbrowser.open_new("http://stackoverflow.com")
```

Este método es comúnmente ignorado por los navegadores modernos y la URL generalmente se abre en una nueva pestaña. El módulo puede intentar abrir una nueva pestaña usando el método `open_new_tab` :

```
import webbrowser
webbrowser.open_new_tab("http://stackoverflow.com")
```

Abrir una URL con diferentes navegadores

El módulo del navegador web también admite diferentes navegadores que utilizan los métodos `register()` y `get()` . El método de obtención se usa para crear un controlador de navegador utilizando una ruta de archivo ejecutable específica y el método de registro se utiliza para adjuntar estos ejecutables a los tipos de navegador predeterminados para su uso futuro, generalmente cuando se usan múltiples tipos de navegador.

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

Registro de un tipo de navegador:

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
webbrowser.register('firefox', None, ff)
# Now to refer to use Firefox in the future you can use this
webbrowser.get('firefox').open("https://stackoverflow.com/")
```

Lea Módulo de navegador web en línea: https://riptutorial.com/es/python/topic/8676/modulo-de-navegador-web

Capítulo 135: Módulo Deque

Sintaxis

- dq = deque () # Crea un deque vacío
- dq = deque (iterable) # Crea un deque con algunos elementos
- dq.append (objeto) # Agrega un objeto a la derecha del deque
- dq.appendleft (objeto) # Agrega un objeto a la izquierda del deque
- dq.pop () -> object # Elimina y devuelve el objeto más a la derecha
- dq.popleft () -> object # Elimina y devuelve el objeto más a la izquierda
- dq.extend (iterable) # Agrega algunos elementos a la derecha del deque
- dq.extendleft (iterable) # Agrega algunos elementos a la izquierda del deque

Parámetros

Parámetro	Detalles
`iterable`	Crea el deque con elementos iniciales copiados de otro iterable.
`maxlen`	Limita qué tan grande puede ser el deque, eliminando elementos antiguos a medida que se agregan nuevos.

Observaciones

Esta clase es útil cuando necesita un objeto similar a una lista que permita operaciones rápidas de agregar y abrir desde cualquier lado (el nombre `deque` significa " *cola de doble extremo* ").

Los métodos proporcionados son, de hecho, muy similares, excepto que algunos como el `pop` , el `append` o la `extend` pueden incluir con el sufijo `left` . La estructura de datos de `deque` debería ser preferible a una lista si uno necesita insertar y eliminar elementos con frecuencia en ambos extremos porque permite hacerlo en tiempo constante O (1).

Examples

Uso básico deque

Los principales métodos que son útiles con esta clase son `popleft` y `appendleft`

```
from collections import deque

d = deque([1, 2, 3])
p = d.popleft()         # p = 1, d = deque([2, 3])
d.appendleft(5)         # d = deque([5, 2, 3])
```

límite de tamaño de salida

Use el parámetro `maxlen` mientras crea un deque para limitar el tamaño del deque:

```
from collections import deque
d = deque(maxlen=3)  # only holds 3 items
d.append(1)  # deque([1])
d.append(2)  # deque([1, 2])
d.append(3)  # deque([1, 2, 3])
d.append(4)  # deque([2, 3, 4]) (1 is removed because its maxlen is 3)
```

Métodos disponibles en deque.

Creando deque vacío:

```
dl = deque()  # deque([]) creating empty deque
```

Creando deque con algunos elementos:

```
dl = deque([1, 2, 3, 4])  # deque([1, 2, 3, 4])
```

Añadiendo elemento a deque:

```
dl.append(5)  # deque([1, 2, 3, 4, 5])
```

Añadiendo elemento del lado izquierdo del deque:

```
dl.appendleft(0)  # deque([0, 1, 2, 3, 4, 5])
```

Añadiendo lista de elementos a deque:

```
dl.extend([6, 7])  # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

Añadiendo lista de elementos desde el lado izquierdo:

```
dl.extendleft([-2, -1])  # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

El uso del elemento `.pop()` eliminará naturalmente un elemento del lado derecho:

```
dl.pop()  # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

Usando el elemento `.popleft()` para eliminar un elemento del lado izquierdo:

```
dl.popleft()  # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

Eliminar elemento por su valor:

```
dl.remove(1)  # deque([-2, 0, 2, 3, 4, 5, 6])
```

Invertir el orden de los elementos en deque:

```
dl.reverse()  # deque([6, 5, 4, 3, 2, 0, -2])
```

Amplia primera búsqueda

Deque es la única estructura de datos de Python con operaciones rápidas de cola . (Tenga en cuenta `queue.Queue` normalmente no es adecuado, ya que está destinado a la comunicación entre subprocesos). Un caso de uso básico de una cola es la primera búsqueda de amplitud .

```
from collections import deque

def bfs(graph, root):
    distances = {}
    distances[root] = 0
    q = deque([root])
    while q:
        # The oldest seen (but not yet visited) node will be the left most one.
        current = q.popleft()
        for neighbor in graph[current]:
            if neighbor not in distances:
                distances[neighbor] = distances[current] + 1
                # When we see a new node, we add it to the right side of the queue.
                q.append(neighbor)
    return distances
```

Digamos que tenemos un gráfico dirigido simple:

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

Ahora podemos encontrar las distancias desde alguna posición inicial:

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

Lea Módulo Deque en línea: https://riptutorial.com/es/python/topic/1976/modulo-deque

Capítulo 136: Módulo Itertools

Sintaxis

- `import itertools`

Examples

Agrupando elementos de un objeto iterable usando una función

Comenzar con un iterable que necesita ser agrupado.

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

Genere el generador agrupado, agrupando por el segundo elemento en cada tupla:

```
def testGroupBy(lst):
    groups = itertools.groupby(lst, key=lambda x: x[1])
    for key, group in groups:
        print(key, list(group))

testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
```

Sólo se agrupan grupos de elementos consecutivos. Es posible que deba ordenar por la misma clave antes de llamar a groupby For Eg, (el último elemento ha cambiado)

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 5, 6)]
testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5)]
# 5 [('c', 5, 6)]
```

El grupo devuelto por groupby es un iterador que no será válido antes de la próxima iteración. Por ejemplo, lo siguiente no funcionará si desea que los grupos se ordenen por clave. El grupo 5 está vacío a continuación porque cuando se busca el grupo 2 invalida 5

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted(groups):
    print(key, list(group))

# 2 [('c', 2, 6)]
# 5 []
```

Para realizar correctamente la clasificación, cree una lista desde el iterador antes de ordenar

```
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted((key, list(group)) for key, group in groups):
    print(key, list(group))

# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
# 5 [('a', 5, 6)]
```

Toma una rebanada de un generador

Itertools "islice" le permite cortar un generador:

```
results = fetch_paged_results()  # returns a generator
limit = 20  # Only want the first 20 results
for data in itertools.islice(results, limit):
    print(data)
```

Normalmente no se puede cortar un generador:

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[:3]:
    print(part)
```

Daré

```
Traceback (most recent call last):
  File "gen.py", line 6, in <module>
    for part in gen()[:3]:
TypeError: 'generator' object is not subscriptable
```

Sin embargo, esto funciona:

```
import itertools

def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in itertools.islice(gen(), 3):
    print(part)
```

Tenga en cuenta que al igual que una división normal, también puede usar los argumentos de `start` , `stop` y `step` :

```
itertools.islice(iterable, 1, 30, 3)
```

itertools.product

Esta función le permite recorrer el producto cartesiano de una lista de iterables.

Por ejemplo,

```
for x, y in itertools.product(xrange(10), xrange(10)):
    print x, y
```

es equivalente a

```
for x in xrange(10):
    for y in xrange(10):
        print x, y
```

Como todas las funciones de python que aceptan un número variable de argumentos, podemos pasar una lista a itertools.product para desempaquetar, con el operador *.

Así,

```
its = [xrange(10)] * 2
for x,y in itertools.product(*its):
    print x, y
```

produce los mismos resultados que los dos ejemplos anteriores.

```
>>> from itertools import product
>>> a=[1,2,3,4]
>>> b=['a','b','c']
>>> product(a,b)
<itertools.product object at 0x0000000002712F78>
>>> for i in product(a,b):
...     print i
...
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
(4, 'a')
(4, 'b')
(4, 'c')
```

itertools.count

Introducción:

Esta simple función genera infinitas series de números. Por ejemplo...

```
for number in itertools.count():
    if number > 20:
        break
    print(number)
```

Tenga en cuenta que hay que romper o se imprime para siempre!

Salida:

```
0
1
2
3
4
5
6
7
8
9
10
```

Argumentos:

`count()` toma dos argumentos, `start` y `step` :

```
for number in itertools.count(start=10, step=4):
    print(number)
    if number > 20:
        break
```

Salida:

```
10
14
18
22
```

itertools.takewhile

itertools.takewhile le permite tomar elementos de una secuencia hasta que una condición se convierte en `False` .

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.takewhile(is_even, lst))

print(result)
```

Esto produce `[0, 2, 4, 12, 18]` .

Tenga en cuenta que, el primer número que viola el predicado (es decir, la función que devuelve un valor booleano) `is_even` is, `13` . Una vez que `takewhile` encuentra un valor que produce `False` para el predicado dado, se rompe.

La **salida producida** por `takewhile` es similar a la salida generada a partir del código siguiente.

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Nota: La concatenación de los resultados producidos por `takewhile` y `dropwhile` produce el iterable original.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

itertools.dropwhile

itertools.dropwhile le permite tomar elementos de una secuencia después de que una condición se convierte en `False` .

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.dropwhile(is_even, lst))

print(result)
```

Esto da como resultado `[13, 14, 22, 23, 44]` .

(*Este ejemplo es el mismo que el de* `takewhile` *pero usando* `dropwhile`) .

Tenga en cuenta que, el primer número que viola el predicado (es decir, la función que devuelve un valor booleano) `is_even` is, `13` . Todos los elementos antes de eso, se descartan.

La **salida producida** por `dropwhile` es similar a la salida generada a partir del código a continuación.

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

La concatenación de los resultados producidos por `takewhile` y `dropwhile` produce el iterable

original.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

Zipping dos iteradores hasta que ambos están agotados

Similar a la función incorporada `zip()` , `itertools.zip_longest` continuará iterando más allá del final del más corto de los dos iterables.

```
from itertools import zip_longest
a = [i for i in range(5)] # Length is 5
b = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # Length is 7
for i in zip_longest(a, b):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

Se puede pasar un argumento de valor de `fillvalue` opcional (predeterminado a `''`) de la siguiente manera:

```
for i in zip_longest(a, b, fillvalue='Hogwash!'):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

En Python 2.6 y 2.7, esta función se llama `itertools.izip_longest` .

Método de combinaciones en el módulo Itertools

`itertools.combinations` devolverá un generador de la secuencia de combinación *k* de una lista.

En otras palabras: devolverá un generador de tuplas de todas las combinaciones posibles de k de la lista de entrada.

Por ejemplo:

Si tienes una lista:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 2))
print b
```

Salida:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

La salida anterior es un generador convertido a una lista de tuplas de todas las combinaciones posibles de *pares* de la lista de entrada `a`

También puedes encontrar todas las 3 combinaciones:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 3))
```

```
print b
```

Salida:

```
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4),
 (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5),
 (2, 4, 5), (3, 4, 5)]
```

Encadenando múltiples iteradores juntos

Use `itertools.chain` para crear un solo generador que produzca los valores de varios generadores en secuencia.

```
from itertools import chain
a = (x for x in ['1', '2', '3', '4'])
b = (x for x in ['x', 'y', 'z'])
' '.join(chain(a, b))
```

Resultados en:

```
'1 2 3 4 x y z'
```

Como constructor alternativo, puede usar el método `chain.from_iterable` que toma como único parámetro un iterable de iterables. Para obtener el mismo resultado que arriba:

```
' '.join(chain.from_iterable([a,b])
```

Mientras que `chain` puede tomar un número arbitrario de argumentos, `chain.from_iterable` es la única manera de encadenar un número *infinito* de iterables.

itertools.repeat

Repetir algo n veces:

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...     print(i)
over-and-over
over-and-over
over-and-over
```

Obtener una suma acumulada de números en un iterable

Python 3.x 3.2

`accumulate` rendimientos una suma acumulada (o producto) de números.

```
>>> import itertools as it
>>> import operator
```

```
>>> list(it.accumulate([1,2,3,4,5]))
[1, 3, 6, 10, 15]

>>> list(it.accumulate([1,2,3,4,5], func=operator.mul))
[1, 2, 6, 24, 120]
```

Recorre los elementos en un iterador

`cycle` es un iterador infinito.

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

Por lo tanto, tenga cuidado de dar límites al usar esto para evitar un bucle infinito. Ejemplo:

```
>>> # Iterate over each element in cycle for a fixed range
>>> cycle_iterator = it.cycle('abc123')
>>> [next(cycle_iterator) for i in range(0, 10)]
['a', 'b', 'c', '1', '2', '3', 'a', 'b', 'c', '1']
```

itertools.permutaciones

`itertools.permutations` devuelve un generador con permutaciones sucesivas de longitud r de elementos en el iterable.

```
a = [1,2,3]
list(itertools.permutations(a))
# [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]

list(itertools.permutations(a, 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

Si la lista `a` tiene elementos duplicados, las permutaciones resultantes tendrán elementos duplicados, puede usar `set` para obtener permutaciones únicas:

```
a = [1,2,1]
list(itertools.permutations(a))
# [(1, 2, 1), (1, 1, 2), (2, 1, 1), (2, 1, 1), (1, 1, 2), (1, 2, 1)]

set(itertools.permutations(a))
# {(1, 1, 2), (1, 2, 1), (2, 1, 1)}
```

Lea Módulo Itertools en línea: https://riptutorial.com/es/python/topic/1564/modulo-itertools

Capítulo 137: Módulo JSON

Observaciones

Para obtener la documentación completa, incluida la funcionalidad específica de la versión, consulte la documentación oficial .

Los tipos

Valores predeterminados

El módulo `json` manejará la codificación y decodificación de los siguientes tipos de manera predeterminada:

Tipos de serialización:

JSON	Pitón
objeto	dictado
formación	lista
cuerda	str
número (int)	En t
numero (real)	flotador
verdadero Falso	Verdadero Falso
nulo	Ninguna

El módulo `json` también entiende `NaN` , `Infinity` e `-Infinity` como sus valores flotantes correspondientes, que están fuera de la especificación JSON.

Tipos de serialización:

Pitón	JSON
dictado	objeto
lista, tupla	formación
str	cuerda

Pitón	JSON
Enums, float, (int / float) -derived	número
Cierto	cierto
Falso	falso
Ninguna	nulo

Para no permitir la codificación de `NaN` , `Infinity` e `-Infinity` , debe codificar con `allow_nan=False` . Esto generará un `ValueError` si intenta codificar estos valores.

Personalización (des) serialización

Existen varios enlaces que le permiten manejar datos que deben representarse de manera diferente. El uso de `functools.partial` permite aplicar parcialmente los parámetros relevantes a estas funciones para su comodidad.

Publicación por entregas:

Puede proporcionar una función que opere en objetos antes de que se serialicen así:

```
# my_json module

import json
from functools import partial

def serialise_object(obj):
    # Do something to produce json-serialisable data
    return dict_obj

dump = partial(json.dump, default=serialise_object)
dumps = partial(json.dumps, default=serialise_object)
```

De serialización:

Hay varios enlaces que son manejados por las funciones json, como object_hook y parse_float. Para obtener una lista exhaustiva de su versión de python, consulte aquí .

```
# my_json module

import json
from functools import partial

def deserialise_object(dict_obj):
    # Do something custom
    return obj

def deserialise_float(str_obj):
    # Do something custom
    return obj
```

```
load = partial(json.load, object_hook=deserialise_object, parse_float=deserialise_float)
loads = partial(json.loads, object_hook=deserialise_object, parse_float=deserialise_float)
```

Mayor (des) serialización personalizada:

El módulo `json` también permite la extensión / sustitución de `json.JSONEncoder` y `json.JSONDecoder` para manejar varios tipos. Los ganchos documentados anteriormente se pueden agregar como valores predeterminados creando un método de nombre equivalente. Para usarlos, simplemente pase la clase como el parámetro `cls` a la función relevante. El uso de `functools.partial` permite aplicar parcialmente el parámetro cls a estas funciones por conveniencia, por ejemplo,

```
# my_json module

import json
from functools import partial

class MyEncoder(json.JSONEncoder):
    # Do something custom

class MyDecoder(json.JSONDecoder):
    # Do something custom

dump = partial(json.dump, cls=MyEncoder)
dumps = partial(json.dumps, cls=MyEncoder)
load = partial(json.load, cls=MyDecoder)
loads = partial(json.loads, cls=MyDecoder)
```

Examples

Creando JSON desde el dictado de Python

```
import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)
```

El fragmento de código anterior devolverá lo siguiente:

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

Creando el dictado de Python desde JSON

```
import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)
```

El fragmento de código anterior devolverá lo siguiente:

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

Almacenamiento de datos en un archivo

El siguiente fragmento de código codifica los datos almacenados en `d` en JSON y los almacena en un archivo (reemplace el `filename` con el nombre real del archivo).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

Recuperando datos de un archivo

El siguiente fragmento de código abre un archivo codificado JSON (reemplaza el `filename` con el nombre real del archivo) y devuelve el objeto que está almacenado en el archivo.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

`load` vs` loads`, `dump` vs` dumps`

El módulo `json` contiene funciones para leer y escribir en y desde cadenas de Unicode, y para leer y escribir en y desde archivos. Estos se diferencian por una `s` final en el nombre de la función. En estos ejemplos, usamos un objeto StringIO, pero las mismas funciones se aplicarían a cualquier objeto similar a un archivo.

Aquí usamos las funciones basadas en cadenas:

```
import json

data = {u"foo": u"bar", u"baz": []}
json_string = json.dumps(data)
# u'{"foo": "bar", "baz": []}'
json.loads(json_string)
# {u"foo": u"bar", u"baz": []}
```

Y aquí usamos las funciones basadas en archivos:

```
import json

from io import StringIO
```

```
json_file = StringIO()
data = {u"foo": u"bar", u"baz": []}
json.dump(data, json_file)
json_file.seek(0)  # Seek back to the start of the file before reading
json_file_content = json_file.read()
# u'{"foo": "bar", "baz": []}'
json_file.seek(0)  # Seek back to the start of the file before reading
json.load(json_file)
# {u"foo": u"bar", u"baz": []}
```

Como puede ver, la principal diferencia es que al descargar datos json debe pasar el identificador de archivo a la función, en lugar de capturar el valor de retorno. También vale la pena señalar que debe buscar el inicio del archivo antes de leer o escribir, para evitar la corrupción de datos. Al abrir un archivo, el cursor se coloca en la posición `0` , por lo que lo siguiente también funcionaría:

```
import json

json_file_path = './data.json'
data = {u"foo": u"bar", u"baz": []}

with open(json_file_path, 'w') as json_file:
    json.dump(data, json_file)

with open(json_file_path) as json_file:
    json_file_content = json_file.read()
    # u'{"foo": "bar", "baz": []}'

with open(json_file_path) as json_file:
    json.load(json_file)
    # {u"foo": u"bar", u"baz": []}
```

Tener las dos formas de tratar con los datos json le permite trabajar de manera idiomática y eficiente con los formatos que se basan en json, como json-per-line de `pyspark` :

```
# loading from a file
data = [json.loads(line) for line in open(file_path).splitlines()]

# dumping to a file
with open(file_path, 'w') as json_file:
    for item in data:
        json.dump(item, json_file)
        json_file.write('\n')
```

Llamando a `json.tool` desde la línea de comandos a la salida JSON de impresión bonita

Dado un archivo JSON "foo.json" como:

```
{"foo": {"bar": {"baz": 1}}}
```

podemos llamar al módulo directamente desde la línea de comando (pasando el nombre del archivo como un argumento) para imprimirlo en forma bonita:

```
$ python -m json.tool foo.json
{
    "foo": {
        "bar": {
            "baz": 1
        }
    }
}
```

El módulo también recibirá información de STDOUT, por lo que (en Bash) también podríamos hacer:

```
$ cat foo.json | python -m json.tool
```

Formato de salida JSON

Digamos que tenemos los siguientes datos:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color":
"black"}]}
```

Simplemente descartando esto como JSON no hace nada especial aquí:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Configuración de sangría para obtener una salida más bonita

Si queremos una impresión bonita, podemos establecer un tamaño de `indent` :

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```

Ordenando las teclas alfabéticamente para

obtener un resultado consistente

Por defecto, el orden de las teclas en la salida no está definido. Podemos obtenerlos en orden alfabético para asegurarnos de que siempre obtengamos la misma salida:

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

Deshacerse de los espacios en blanco para obtener una salida compacta

Podríamos querer deshacernos de los espacios innecesarios, lo que se hace configurando cadenas separadoras diferentes de las predeterminadas `', '` y `': '` :

```
>>>print(json.dumps(data, separators=(',', ':')))
{"cats":[{"name":"Tubbs","color":"white"},{"name":"Pepper","color":"black"}]}
```

JSON que codifica objetos personalizados

Si solo intentamos lo siguiente:

```
import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))
```

recibimos un error que dice que `TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializable` .

Para poder serializar correctamente el objeto de fecha y hora, necesitamos escribir un código personalizado sobre cómo convertirlo:

```
class DatetimeJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        try:
            return obj.isoformat()
        except AttributeError:
            # obj has no isoformat method; let the builtin JSON encoder handle it
            return super(DatetimeJSONEncoder, self).default(obj)
```

y luego use esta clase de codificador en lugar de `json.dumps` :

```
encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# prints {"datetime": "2016-09-26T04:44:00"}
```

Lea Módulo JSON en línea: https://riptutorial.com/es/python/topic/272/modulo-json

Capítulo 138: Módulo operador

Examples

Operadores como alternativa a un operador infijo.

Para cada operador de infijo, por ejemplo, `+` hay una función de `operator` (`operator.add` para `+`):

```
1 + 1
# Output: 2
from operator import add
add(1, 1)
# Output: 2
```

aunque la documentación principal indica que para los operadores aritméticos solo se permite la entrada numérica, *es* posible:

```
from operator import mul
mul('a', 10)
# Output: 'aaaaaaaaaa'
mul([3], 3)
# Output: [3, 3, 3]
```

Vea también: asignación de la función de operación a operador en la documentación oficial de Python .

Methodcaller

En lugar de esta función `lambda` que llama explícitamente al método:

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist))     # Keep only elements that start with 'd'
# Output: ['duck']
```

uno podría usar una función de operador que hace lo mismo:

```
from operator import methodcaller
list(filter(methodcaller('startswith', 'd'), alist)) # Does the same but is faster.
# Output: ['duck']
```

Itemgetter

Agrupando los pares clave-valor de un diccionario por el valor con `itemgetter` :

```
from itertools import groupby
from operator import itemgetter
adict = {'a': 1, 'b': 5, 'c': 1}
```

```
dict((i, dict(v)) for i, v in groupby(adict.items(), itemgetter(1)))
# Output: {1: {'a': 1, 'c': 1}, 5: {'b': 5}}
```

que es equivalente (pero más rápido) a una función `lambda` como esta:

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

O clasificando una lista de tuplas por el segundo elemento primero el primer elemento como secundario:

```
alist_of_tuples = [(5,2), (1,3), (2,2)]
sorted(alist_of_tuples, key=itemgetter(1,0))
# Output: [(2, 2), (5, 2), (1, 3)]
```

Lea Módulo operador en línea: https://riptutorial.com/es/python/topic/257/modulo-operador

Capítulo 139: módulo pyautogui

Introducción

pyautogui es un módulo usado para controlar el mouse y el teclado. Este módulo se usa básicamente para automatizar el clic del mouse y las tareas de pulsación del teclado. Para el mouse, las coordenadas de la pantalla (0,0) comienzan desde la esquina superior izquierda. Si está fuera de control, mueva rápidamente el cursor del mouse hacia la parte superior izquierda, tomará el control del mouse y el teclado del Python y se lo devolverá.

Examples

Funciones del mouse

Estas son algunas de las funciones útiles del mouse para controlar el mouse.

```
size()           #gave you the size of the screen
position()     #return current position of mouse
moveTo(200,0,duration=1.5)     #move the cursor  to (200,0) position  with 1.5 second delay

moveRel()          #move the cursor relative to your current position.
click(337,46)           #it will click on the position mention there
dragRel()              #it will drag the mouse relative to position
pyautogui.displayMousePosition()     #gave you the current mouse position but should be done
on terminal.
```

Funciones del teclado

Estas son algunas de las funciones útiles del teclado para automatizar la pulsación de teclas.

```
typewrite('')    #this will type the string on the screen where current window has focused.
typewrite(['a','b','left','left','X','Y'])
pyautogui.KEYBOARD_KEYS    #get the list of all the keyboard_keys.
pyautogui.hotkey('ctrl','o')    #for the combination of keys to enter.
```

ScreenShot y reconocimiento de imágenes

Esta función te ayudará a tomar la captura de pantalla y también a relacionar la imagen con la parte de la pantalla.

```
.screenshot('c:\\path')        #get the screenshot.
.locateOnScreen('c:\\path')    #search that image on screen and get the coordinates for you.
locateCenterOnScreen('c:\\path')       #get the coordinate for the image on screen.
```

Lea módulo pyautogui en línea: https://riptutorial.com/es/python/topic/9432/modulo-pyautogui

Capítulo 140: Módulo Sqlite3

Examples

Sqlite3 - No requiere proceso de servidor separado.

El módulo sqlite3 fue escrito por Gerhard Häring. Para usar el módulo, primero debe crear un objeto de conexión que represente la base de datos. Aquí los datos se almacenarán en el archivo example.db:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

También puede proporcionar el nombre especial: memoria: para crear una base de datos en la RAM. Una vez que tenga una conexión, puede crear un objeto Cursor y llamar a su método execute () para ejecutar comandos SQL:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
         (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

Obtención de los valores de la base de datos y manejo de errores.

Obteniendo los valores de la base de datos SQLite3.

Imprimir valores de fila devueltos por consulta de selección

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute("SELECT * from table_name where id=cust_id")
for row in c:
    print row # will be a list
```

Para obtener un solo método fetchone () coincidente

```
print c.fetchone()
```

Para filas múltiples use el método fetchall ()

```
a=c.fetchall() #which is similar to list(cursor) method used previously
for row in a:
    print row
```

El manejo de errores se puede hacer usando la función incorporada sqlite3.Error

```
try:
    #SQL Code
except sqlite3.Error as e:
    print "An error occurred:", e.args[0]
```

Lea Módulo Sqlite3 en línea: https://riptutorial.com/es/python/topic/7754/modulo-sqlite3

Capítulo 141: Multihilo

Introducción

Los subprocesos permiten que los programas de Python manejen múltiples funciones a la vez en lugar de ejecutar una secuencia de comandos individualmente. Este tema explica los principios detrás de los hilos y demuestra su uso.

Examples

Conceptos básicos de multihilo

Usando el módulo de `threading` , se puede iniciar un nuevo subproceso de ejecución creando un nuevo `threading.Thread` y asigne una función para ejecutar:

```
import threading

def foo():
  print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

El parámetro de `target` referencia a la función (u objeto llamable) que se ejecutará. El subproceso no comenzará la ejecución hasta que se llame al `start` en el objeto `Thread` .

Comenzando un hilo

```
my_thread.start() # prints 'Hello threading!'
```

Ahora que `my_thread` ha ejecutado y finalizado, al `start` nuevo, se producirá un `RuntimeError` . Si desea ejecutar su hilo como un demonio, pasar el `daemon=True` kwarg, o configurar `my_thread.daemon` en `True` antes de llamar a `start()` , hace que su `Thread` ejecute silenciosamente en segundo plano como un demonio.

Unirse a un hilo

En los casos en que divide un trabajo grande en varios pequeños y desea ejecutarlos simultáneamente, pero debe esperar a que todos terminen antes de continuar, `Thread.join()` es el método que está buscando.

Por ejemplo, supongamos que desea descargar varias páginas de un sitio web y compilarlas en una sola página. Tu harias esto

```
import requests
from threading import Thread
from queue import Queue
```

```
q = Queue(maxsize=20)
def put_page_to_q(page_num):
    q.put(requests.get('http://some-website.com/page_%s.html' % page_num)

def compile(q):
    # magic function that needs all pages before being able to be executed
    if not q.full():
        raise ValueError
    else:
        print("Done compiling!")

threads = []
for page_num in range(20):
     t = Thread(target=requests.get, args=(page_num,))
     t.start()
     threads.append(t)

# Next, join all threads to make sure all threads are done running before
# we continue. join() is a blocking call (unless specified otherwise using
# the kwarg blocking=False when calling join)
for t in threads:
    t.join()

# Call compile() now, since all threads have completed
compile(q)
```

Una mirada más cercana a cómo funciona `join()` se puede encontrar aquí .

Crear una clase de hilo personalizado

Usando la clase `threading.Thread` podemos subclasificar la nueva clase Thread personalizada. debemos anular el método de `run` en una subclase.

```
from threading import Thread
import time

class Sleepy(Thread):

    def run(self):
        time.sleep(5)
        print("Hello form Thread")

if __name__ == "__main__":
    t = Sleepy()
    t.start()      # start method automatic call Thread class run method.
    # print 'The main program continues to run in foreground.'
    t.join()
    print("The main program continues to run in the foreground.")
```

Comunicando entre hilos

Hay múltiples hilos en su código y necesita comunicarse de forma segura entre ellos.

Puede utilizar una `Queue` de la biblioteca de `queue` .

```
from queue import Queue
```

```
from threading import Thread

# create a data producer
def producer(output_queue):
    while True:
        data = data_computation()

        output_queue.put(data)

# create a consumer
def consumer(input_queue):
    while True:
        # retrieve data (blocking)
        data = input_queue.get()

        # do something with the data

        # indicate data has been consumed
        input_queue.task_done()
```

Creación de subprocesos de productor y consumidor con una cola compartida

```
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Creando un grupo de trabajadores

Usando `threading` y `queue` :

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_server(addr, nworkers):
    print('Echo server running at', addr)
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

Utilizando `concurrent.futures.Threadpoolexecutor` :

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_server(addr):
    print('Echo server running at', addr)
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

Python Cookbook, 3ra edición, por David Beazley y Brian K. Jones (O'Reilly). Derechos de autor 2013 David Beazley y Brian Jones, 978-1-449-34037-7.

Uso avanzado de multihilos

Esta sección contendrá algunos de los ejemplos más avanzados realizados utilizando Multithreading.

Impresora avanzada (logger)

Un hilo que imprime todo se recibe y modifica la salida de acuerdo con el ancho del terminal. Lo bueno es que también la salida "ya escrita" se modifica cuando cambia el ancho del terminal.

```
#!/usr/bin/env python2

import threading
import Queue
import time
import sys
import subprocess
from backports.shutil_get_terminal_size import get_terminal_size

printq = Queue.Queue()
interrupt = False
lines = []

def main():

    ptt = threading.Thread(target=printer) # Turn the printer on
    ptt.daemon = True
    ptt.start()

    # Stupid example of stuff to print
    for i in xrange(1,100):
        printq.put(' '.join([str(x) for x in range(1,i)]))           # The actual way to send
stuff to the printer
        time.sleep(.5)

def split_line(line, cols):
    if len(line) > cols:
        new_line = ''
```

```
        ww = line.split()
        i = 0
        while len(new_line) <= (cols - len(ww[i]) - 1):
            new_line += ww[i] + ' '
            i += 1
            print len(new_line)
        if new_line == '':
            return (line, '')

        return (new_line, ' '.join(ww[i:]))
    else:
        return (line, '')


def printer():

    while True:
        cols, rows = get_terminal_size() # Get the terminal dimensions
        msg = '#' + '-' * (cols - 2) + '#\n' # Create the
        try:
            new_line = str(printq.get_nowait())
            if new_line != '!@#EXIT#@!': # A nice way to turn the printer
                                         # thread out gracefully
                lines.append(new_line)
                printq.task_done()
            else:
                printq.task_done()
                sys.exit()
        except Queue.Empty:
            pass

        # Build the new message to show and split too long lines
        for line in lines:
            res = line          # The following is to split lines which are
                                # longer than cols.
            while len(res) !=0:
                toprint, res = split_line(res, cols)
                msg += '\n' + toprint

        # Clear the shell and print the new output
        subprocess.check_call('clear') # Keep the shell clean
        sys.stdout.write(msg)
        sys.stdout.flush()
        time.sleep(.5)
```

Hilo que se puede detener con un bucle de tiempo

```
import threading
import time

class StoppableThread(threading.Thread):
    """Thread class with a stop() method. The thread itself has to check
    regularly for the stopped() condition."""

    def __init__(self):
        super(StoppableThread, self).__init__()
        self._stop_event = threading.Event()

    def stop(self):
```

```
        self._stop_event.set()

    def join(self, *args, **kwargs):
        self.stop()
        super(StoppableThread,self).join(*args, **kwargs)

    def run()
        while not self._stop_event.is_set():
            print("Still running!")
            time.sleep(2)
        print("stopped!"
```

Basado en esta pregunta .

Lea Multihilo en línea: https://riptutorial.com/es/python/topic/544/multihilo

Capítulo 142: Multiprocesamiento

Examples

Ejecutando dos procesos simples

Un ejemplo simple de usar múltiples procesos serían dos procesos (trabajadores) que se ejecutan por separado. En el siguiente ejemplo, se inician dos procesos:

- `countUp()` cuenta 1 arriba, cada segundo.
- `countDown()` cuenta 1 abajo, cada segundo.

```
import multiprocessing
import time
from random import randint

def countUp():
    i = 0
    while i <= 3:
        print('Up:\t{}'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i += 1

def countDown():
    i = 3
    while i >= 0:
        print('Down:\t{}'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i -= 1

if __name__ == '__main__':
    # Initiate the workers.
    workerUp = multiprocessing.Process(target=countUp)
    workerDown = multiprocessing.Process(target=countDown)

    # Start the workers.
    workerUp.start()
    workerDown.start()

    # Join the workers. This will block in the main (parent) process
    # until the workers are complete.
    workerUp.join()
    workerDown.join()
```

La salida es la siguiente:

```
Up:    0
Down:    3
Up:    1
Up:    2
Down:    2
Up:    3
Down:    1
Down:    0
```

Uso de la piscina y el mapa

```
from multiprocessing import Pool

def cube(x):
    return x ** 3

if __name__ == "__main__":
    pool = Pool(5)
    result = pool.map(cube, [0, 1, 2, 3])
```

`Pool` es una clase que administra varios `Workers` (procesos) detrás de escena y le permite al programador usar.

`Pool(5)` crea un nuevo pool con 5 procesos, y `pool.map` funciona igual que el mapa, pero usa varios procesos (la cantidad definida al crear el pool).

Se pueden obtener resultados similares utilizando `map_async` , `apply` y `apply_async` que se pueden encontrar en la documentación .

Lea Multiprocesamiento en línea: https://riptutorial.com/es/python/topic/3601/multiprocesamiento

Capítulo 143: Mutable vs Inmutable (y Hashable) en Python

Examples

Mutable vs inmutable

Hay dos tipos de tipos en Python. Tipos inmutables y tipos mutables.

Inmutables

Un objeto de un tipo inmutable no puede ser cambiado. Cualquier intento de modificar el objeto dará lugar a que se cree una copia.

Esta categoría incluye: enteros, flotadores, complejos, cadenas, bytes, tuplas, rangos y conjuntos de imágenes.

Para resaltar esta propiedad, vamos a jugar con el `id` incorporado. Esta función devuelve el identificador único del objeto pasado como parámetro. Si el id es el mismo, este es el mismo objeto. Si cambia, entonces este es otro objeto. *(Algunos dicen que esta es realmente la dirección de memoria del objeto, pero ten cuidado con ellos, son del lado oscuro de la fuerza ...)*

```
>>> a = 1
>>> id(a)
140128142243264
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

Está bien, 1 no es 3 ... Noticias de última hora ... Tal vez no. Sin embargo, este comportamiento a menudo se olvida cuando se trata de tipos más complejos, especialmente de cadenas.

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
140128123955504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

Jajaja ¿Ver? ¡Podemos modificarlo!

```
>>> id(stack)
140128123911472
```

No. Si bien parece que podemos cambiar la cadena nombrada por la `stack` variables, lo que realmente hacemos es crear un nuevo objeto para contener el resultado de la concatenación. Nos engañan porque en el proceso, el objeto antiguo no va a ninguna parte, por lo que se destruye. En otra situación, eso habría sido más obvio:

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

En este caso, está claro que si queremos conservar la primera cadena, necesitamos una copia. ¿Pero es eso tan obvio para otros tipos?

Ejercicio

Ahora, sabiendo cómo funcionan los tipos inmutables, ¿qué diría usted con el siguiente código? ¿Es sabio?

```
s = ""
for i in range(1, 1000):
    s += str(i)
    s += ","
```

Mutables

Un objeto de un tipo mutable se puede cambiar y se cambia *in situ* . No se realizan copias implícitas.

Esta categoría incluye: listas, diccionarios, bytearrays y sets.

Sigamos jugando con nuestra pequeña función de `id` .

```
>>> b = bytearray(b'Stack')
>>> b
bytearray(b'Stack')
>>> b = bytearray(b'Stack')
>>> id(b)
140128030688288
>>> b += b'Overflow'
>>> b
bytearray(b'StackOverflow')
>>> id(b)
140128030688288
```

(Como nota al margen, uso bytes que contienen datos ASCII para aclarar mi punto, pero recuerde que los bytes no están diseñados para contener datos textuales. Que la fuerza me perdone).

¿Que tenemos? Creamos un bytearray, lo modificamos y usando el `id` , podemos asegurarnos de

que este es el mismo objeto, modificado. No es una copia de eso.

Por supuesto, si un objeto se va a modificar con frecuencia, un tipo mutable hace un trabajo mucho mejor que un tipo inmutable. Desafortunadamente, la realidad de esta propiedad a menudo se olvida cuando más duele.

```
>>> c = b
>>> c += b' rocks!'
>>> c
bytearray(b'StackOverflow rocks!')
```

Bueno...

```
>>> b
bytearray(b'StackOverflow rocks!')
```

Waiiit un segundo ...

```
>>> id(c) == id(b)
True
```

En efecto. `c` no es una copia de `b` . `c` es `b` .

Ejercicio

Ahora que entiendes mejor qué efecto secundario implica un tipo mutable, ¿puedes explicar qué está mal en este ejemplo?

```
>>> ll = [ [] ]*4 # Create a list of 4 lists to contain our results
>>> ll
[[], [], [], []]
>>> ll[0].append(23) # Add result 23 to first list
>>> ll
[[23], [23], [23], [23]]
>>> # Oops...
```

Mutables e inmutables como argumentos

Uno de los principales casos de uso cuando un desarrollador necesita tener en cuenta la mutabilidad es cuando pasa argumentos a una función. Esto es muy importante, ya que esto determinará la capacidad de la función para modificar objetos que no pertenecen a su alcance, o en otras palabras, si la función tiene efectos secundarios. Esto también es importante para comprender dónde debe estar disponible el resultado de una función.

```
>>> def list_add3(lin):
    lin += [3]
    return lin

>>> a = [1, 2, 3]
>>> b = list_add3(a)
```

```
>>> b
[1, 2, 3, 3]
>>> a
[1, 2, 3, 3]
```

Aquí, el error es pensar que `lin` , como parámetro de la función, puede modificarse localmente. En su lugar, `lin` y `a` referencia del mismo objeto. Como este objeto es mutable, la modificación se realiza in situ, lo que significa que el objeto al que hacen referencia tanto `lin` como `a` se modifica. No es necesario devolver `lin` , porque ya tenemos una referencia a este objeto en forma de `a` . `a` y `b` terminan haciendo referencia al mismo objeto.

Esto no es lo mismo para las tuplas.

```
>>> def tuple_add3(tin):
    tin += (3,)
    return tin

>>> a = (1, 2, 3)
>>> b = tuple_add3(a)
>>> b
(1, 2, 3, 3)
>>> a
(1, 2, 3)
```

Al comienzo de la función, `tin` y `a` referencia del mismo objeto. Pero este es un objeto inmutable. Así que cuando la función intenta modificarla, `tin` recibir un nuevo objeto con la modificación, mientras que `a` mantiene una referencia al objeto original. En este caso, devolver `tin` es obligatorio, o el nuevo objeto se perdería.

Ejercicio

```
>>> def yoda(prologue, sentence):
    sentence.reverse()
    prologue += " ".join(sentence)
    return prologue

>>> focused = ["You must", "stay focused"]
>>> saying = "Yoda said: "
>>> yoda_sentence = yoda(saying, focused)
```

Nota: el `reverse` opera en el lugar.

¿Qué opinas de esta función? ¿Tiene efectos secundarios? ¿Es necesaria la devolución? Después de la llamada, ¿cuál es el valor de `saying` ? De `focused` ? ¿Qué sucede si se vuelve a llamar a la función con los mismos parámetros?

Lea Mutable vs Inmutable (y Hashable) en Python en línea:
https://riptutorial.com/es/python/topic/9182/mutable-vs-inmutable--y-hashable--en-python

Capítulo 144: Neo4j y Cypher usando Py2Neo

Examples

Importación y Autenticación

```
from py2neo import authenticate, Graph, Node, Relationship
authenticate("localhost:7474", "neo4j", "<pass>")
graph = Graph()
```

Debe asegurarse de que su base de datos Neo4j existe en localhost: 7474 con las credenciales apropiadas.

El objeto `graph` es su interfaz con la instancia de neo4j en el resto de su código de Python. Más bien, gracias a hacer de esto una variable global, debes mantenerla en el método `__init__` una clase.

Añadiendo nodos a Neo4j Graph

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    article.properties["title"] = results[r]['news_title']
    article.properties["timestamp"] = results[r]['news_timestamp']
    article.push()
    [...]
```

Agregar nodos a la gráfica es bastante simple, `graph.merge_one` es importante ya que evita elementos duplicados. (Si ejecuta el script dos veces, la segunda vez se actualizará el título y no se crearán nuevos nodos para los mismos artículos)

`timestamp` debe ser un número entero y no una cadena de fecha, ya que neo4j realmente no tiene un tipo de datos de fecha. Esto causa problemas de clasificación cuando almacena la fecha como '05 -06-1989 '

`article.push()` es una llamada que realmente realiza la operación en neo4j. No olvides este paso.

Agregando relaciones a Neo4j Graph

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    if 'LOCATION' in results[r].keys():
        for loc in results[r]['LOCATION']:
            loc = graph.merge_one("Location", "name", loc)
            try:
                rel = graph.create_unique(Relationship(article, "about_place", loc))
            except Exception, e:
                print e
```

`create_unique` es importante para evitar duplicados. Pero por lo demás es una operación bastante sencilla. El nombre de la relación también es importante, ya que lo usaría en casos avanzados.

Consulta 1: Autocompletar en títulos de noticias

```
def get_autocomplete(text):
    query = """
    start n = node(*) where n.name =~ '(?i)%s.*' return n.name,labels(n) limit 10;
    """
    query = query % (text)
    obj = []
    for res in graph.cypher.execute(query):
        # print res[0],res[1]
        obj.append({'name':res[0],'entity_type':res[1]})
    return res
```

Esta es una consulta de ejemplo para obtener todos los nodos con el `name` la propiedad que comienza con el `text` argumento.

Consulta 2: obtener artículos de noticias por ubicación en una fecha en particular

```
def search_news_by_entity(location,timestamp):
    query = """
    MATCH (n)-[]->(l)
    where l.name='%s' and n.timestamp='%s'
    RETURN n.news_id limit 10
    """

    query = query % (location,timestamp)

    news_ids = []
    for res in graph.cypher.execute(query):
        news_ids.append(str(res[0]))

    return news_ids
```

Puede usar esta consulta para encontrar todos los artículos de noticias `(n)` conectados a una ubicación `(l)` por una relación.

Cypher Query Samples

Contar artículos conectados a una persona en particular a lo largo del tiempo.

```
MATCH (n)-[]->(l)
where l.name='Donald Trump'
RETURN n.date,count(*) order by n.date
```

Busque otras personas / ubicaciones conectadas a los mismos artículos de noticias que Trump con al menos 5 nodos de relaciones totales.

```
MATCH (n:NewsArticle)-[]->(l)
```

```
where l.name='Donald Trump'
MATCH (n:NewsArticle)-[]->(m)
with m,count(n) as num where num>5
return labels(m)[0],(m.name), num order by num desc limit 10
```

Lea Neo4j y Cypher usando Py2Neo en línea: https://riptutorial.com/es/python/topic/5841/neo4j-y-cypher-usando-py2neo

Capítulo 145: Nodo de lista enlazada

Examples

Escribe un nodo de lista enlazada simple en python

Una lista enlazada es:

- la lista vacía, representada por Ninguna, o
- un nodo que contiene un objeto de carga y una referencia a una lista enlazada.

```
#! /usr/bin/env python

class Node:
      def __init__(self, cargo=None, next=None):
          self.car = cargo
          self.cdr = next
      def __str__(self):
          return str(self.car)


      def display(lst):
          if lst:
             w("%s " % lst)
             display(lst.cdr)
          else:
             w("nil\n")
```

Lea Nodo de lista enlazada en línea: https://riptutorial.com/es/python/topic/6916/nodo-de-lista-enlazada

Capítulo 146: Objetos de propiedad

Observaciones

Nota : en Python 2, asegúrese de que su clase herede del objeto (lo que lo convierte en una clase de nuevo estilo) para que todas las características de las propiedades estén disponibles.

Examples

Usando el decorador @property

El decorador de `@property` se puede utilizar para definir métodos en una clase que actúan como atributos. Un ejemplo en el que esto puede ser útil es cuando se expone información que puede requerir una búsqueda inicial (costosa) y una recuperación sencilla a partir de entonces.

Dado algún módulo de `foobar.py` :

```
class Foo(object):
    def __init__(self):
        self.__bar = None

    @property
    def bar(self):
        if self.__bar is None:
            self.__bar = some_expensive_lookup_operation()
        return self.__bar
```

Entonces

```
>>> from foobar import Foo
>>> foo = Foo()
>>> print(foo.bar)  # This will take some time since bar is None after initialization
42
>>> print(foo.bar)  # This is much faster since bar has a value now
42
```

Usando el decorador de propiedad para las propiedades de lectura-escritura

Si desea usar `@property` para implementar un comportamiento personalizado para configurar y obtener, use este patrón:

```
class Cash(object):
    def __init__(self, value):
        self.value = value
    @property
    def formatted(self):
        return '${:.2f}'.format(self.value)
    @formatted.setter
```

```
    def formatted(self, new):
        self.value = float(new[1:])
```

Para usar esto:

```
>>> wallet = Cash(2.50)
>>> print(wallet.formatted)
$2.50
>>> print(wallet.value)
2.5
>>> wallet.formatted = '$123.45'
>>> print(wallet.formatted)
$123.45
>>> print(wallet.value)
123.45
```

Anulando solo un captador, configurador o un eliminador de un objeto de propiedad

Cuando se hereda de una clase con una propiedad, puede proporcionar una nueva aplicación para una o más de las propiedades `getter` , `setter` o `deleter` funciones, haciendo referencia a la propiedad objeto *de la clase padre:*

```
class BaseClass(object):
    @property
    def foo(self):
        return some_calculated_value()

    @foo.setter
    def foo(self, value):
        do_something_with_value(value)


class DerivedClass(BaseClass):
    @BaseClass.foo.setter
    def foo(self, value):
        do_something_different_with_value(value)
```

También puede agregar un definidor o un eliminador donde antes no había uno en la clase base.

Usando propiedades sin decoradores

Si bien el uso de la sintaxis decorativa (con la @) es conveniente, también es un poco oculto. Puede utilizar propiedades directamente, sin decoradores. El siguiente ejemplo de Python 3.x muestra esto:

```
class A:
    p = 1234
    def getX (self):
        return self._x

    def setX (self, value):
        self._x = value
```

```
    def getY (self):
        return self._y

    def setY (self, value):
        self._y = 1000 + value    # Weird but possible

    def getY2 (self):
        return self._y

    def setY2 (self, value):
        self._y = value

    def getT    (self):
        return self._t

    def setT (self, value):
        self._t = value

    def getU (self):
        return self._u + 10000

    def setU (self, value):
        self._u = value - 5000

    x, y, y2 = property (getX, setX), property (getY, setY), property (getY2, setY2)
    t = property (getT, setT)
    u = property (getU, setU)

A.q = 5678

class B:
    def getZ (self):
        return self.z_

    def setZ (self, value):
        self.z_ = value

    z = property (getZ, setZ)

class C:
    def __init__ (self):
        self.offset = 1234

    def getW (self):
        return self.w_ + self.offset

    def setW (self, value):
        self.w_ = value - self.offset

    w = property (getW, setW)

a1 = A ()
a2 = A ()

a1.y2 = 1000
a2.y2 = 2000

a1.x = 5
a1.y = 6
```

```
a2.x = 7
a2.y = 8

a1.t = 77
a1.u = 88

print (a1.x, a1.y, a1.y2)
print (a2.x, a2.y, a2.y2)
print (a1.p, a2.p, a1.q, a2.q)

print (a1.t, a1.u)

b = B ()
c = C ()

b.z = 100100
c.z = 200200
c.w = 300300

print (a1.x, b.z, c.z, c.w)

c.w = 400400
c.z = 500500
b.z = 600600

print (a1.x, b.z, c.z, c.w)
```

Lea Objetos de propiedad en línea: https://riptutorial.com/es/python/topic/2050/objetos-de-propiedad

Capítulo 147: Operadores booleanos

Examples

y

Evalúa el segundo argumento si y solo si ambos argumentos son veraces. De lo contrario se evalúa al primer argumento falsey.

```
x = True
y = True
z = x and y # z = True

x = True
y = False
z = x and y # z = False

x = False
y = True
z = x and y # z = False

x = False
y = False
z = x and y # z = False

x = 1
y = 1
z = x and y # z = y, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 0
y = 1
z = x and y # z = x, so z = 0 (see above)

x = 1
y = 0
z = x and y # z = y, so z = 0 (see above)

x = 0
y = 0
z = x and y # z = x, so z = 0 (see above)
```

Los `1` en el ejemplo anterior se pueden cambiar a cualquier valor verdadero, y los `0` se pueden cambiar a cualquier valor falso.

o

Evalúa el primer argumento de verdad si alguno de los argumentos es verdadero. Si ambos argumentos son falsey, evalúa el segundo argumento.

```
x = True
y = True
z = x or y # z = True
```

```
x = True
y = False
z = x or y # z = True

x = False
y = True
z = x or y # z = True

x = False
y = False
z = x or y # z = False

x = 1
y = 1
z = x or y # z = x, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 1
y = 0
z = x or y # z = x, so z = 1 (see above)

x = 0
y = 1
z = x or y # z = y, so z = 1 (see above)

x = 0
y = 0
z = x or y # z = y, so z = 0 (see above)
```

Los `1` en el ejemplo anterior se pueden cambiar a cualquier valor verdadero, y los `0` se pueden cambiar a cualquier valor falso.

no

Devuelve lo contrario de la siguiente declaración:

```
x = True
y = not x # y = False

x = False
y = not x # y = True
```

Evaluación de cortocircuito

Python evalúa mínimamente las expresiones booleanas.

```
>>> def true_func():
...     print("true_func()")
...     return True
...
>>> def false_func():
...     print("false_func()")
...     return False
...
>>> true_func() or false_func()
true_func()
True
```

```
>>> false_func() or true_func()
false_func()
true_func()
True
>>> true_func() and false_func()
true_func()
false_func()
False
>>> false_func() and false_func()
false_func()
False
```

`and` y` or` no están garantizados para devolver un valor booleano

Cuando usa `or` , devolverá el primer valor en la expresión si es verdadero, de lo contrario, devolverá ciegamente el segundo valor. Es decir `or` es equivalente a:

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

Para `and` , devolverá su primer valor si es falso, de lo contrario, devolverá el último valor:

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

Un simple ejemplo

En Python puedes comparar un solo elemento utilizando dos operadores binarios, uno en cada lado:

```
if 3.14 < x < 3.142:
    print("x is near pi")
```

En muchos (¿la mayoría?) Lenguajes de programación, esto se evaluaría de forma contraria a las matemáticas regulares: `(3.14 < x) < 3.142` , pero en Python se trata como `3.14 < x and x < 3.142` , como la mayoría de los no programadores Esperaría.

Lea Operadores booleanos en línea: https://riptutorial.com/es/python/topic/1731/operadores-booleanos

Capítulo 148: Operadores de Bitwise

Introducción

Las operaciones bitwise alteran cadenas binarias en el nivel de bit. Estas operaciones son increíblemente básicas y están directamente soportadas por el procesador. Estas pocas operaciones son necesarias para trabajar con controladores de dispositivo, gráficos de bajo nivel, criptografía y comunicaciones de red. Esta sección proporciona conocimientos útiles y ejemplos de operadores bitwise de Python.

Sintaxis

- x << y # Bitwise Left Shift
- x >> y # Bitwise Right Shift
- x & y # Bitwise Y
- x | y # Bitwise OR
- ~ x # Bitwise NO
- x ^ y # Bitwise XOR

Examples

Y a nivel de bit

El operador `&` realizará un **AND** binario, donde se copia un bit si existe en **ambos** operandos. Eso significa:

```
# 0 & 0 = 0
# 0 & 1 = 0
# 1 & 0 = 0
# 1 & 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 & 30
# Out: 28
# 28 = 0b11100

bin(60 & 30)
# Out: 0b11100
```

Bitwise o

El `|` el operador realizará un binario "o", donde se copia un bit si existe en cualquiera de los

operandos. Eso significa:

```
# 0 | 0 = 0
# 0 | 1 = 1
# 1 | 0 = 1
# 1 | 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 | 30
# Out: 62
# 62 = 0b111110

bin(60 | 30)
# Out: 0b111110
```

XOR de bitwise (OR exclusivo)

El operador `^` realizará un **XOR** binario en el que se copia un binario `1` si y solo si es el valor de exactamente **un** operando. Otra forma de afirmar esto es que el resultado es `1` solo si los operandos son diferentes. Ejemplos incluyen:

```
# 0 ^ 0 = 0
# 0 ^ 1 = 1
# 1 ^ 0 = 1
# 1 ^ 1 = 0

# 60 = 0b111100
# 30 = 0b011110
60 ^ 30
# Out: 34
# 34 = 0b100010

bin(60 ^ 30)
# Out: 0b100010
```

Desplazamiento a la izquierda en modo de bits

El operador `<<` realizará un "desplazamiento a la izquierda" a nivel de bits, donde el valor del operando izquierdo se mueve a la izquierda por el número de bits dado por el operando derecho.

```
# 2 = 0b10
2 << 2
# Out: 8
# 8 = 0b1000

bin(2 << 2)
# Out: 0b1000
```

Realizar un cambio de bit a la izquierda de `1` es equivalente a la multiplicación por `2` :

```
7 << 1
# Out: 14
```

Realizar un cambio de bit a la izquierda de `n` es equivalente a la multiplicación por `2**n` :

```
3 << 4
# Out: 48
```

Cambio a la derecha en el modo de bits

El operador `>>` realizará un "desplazamiento a la derecha" a nivel de bits, donde el valor del operando izquierdo se mueve a la derecha por el número de bits dado por el operando derecho.

```
# 8 = 0b1000
8 >> 2
# Out: 2
# 2 = 0b10

bin(8 >> 2)
# Out: 0b10
```

Realizar un cambio de bit a la derecha de `1` es equivalente a la división entera por `2` :

```
36 >> 1
# Out: 18

15 >> 1
# Out: 7
```

Realizar un cambio de bit a la derecha de `n` es equivalente a la división entera por `2**n` :

```
48 >> 4
# Out: 3

59 >> 3
# Out: 7
```

Bitwise NO

El operador `~` volteará todos los bits en el número. Dado que las computadoras usan representaciones de números firmados , especialmente la notación de complemento de los dos para codificar números binarios negativos donde los números negativos se escriben con un (1) inicial en lugar de un cero (0).

Esto significa que si estuviera usando 8 bits para representar los números del complemento a dos, trataría los patrones de `0000 0000` a `0111 1111` para representar números de 0 a 127 y reservaría `1xxx xxxx` para representar números negativos.

Números de ocho bits de complemento a dos

Bits	Valor sin firmar	Valor del complemento a dos
0000 0000	0	0

Bits	Valor sin firmar	Valor del complemento a dos
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
1111 1110	254	-2
1111 1111	255	-1

En esencia, esto significa que mientras `1010 0110` tiene un valor sin signo de 166 (se obtiene al agregar `(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)`), tiene un valor de complemento a dos de -90 (se obtiene al agregar `(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)` , y complementando el valor).

De esta manera, los números negativos varían hasta -128 (`1000 0000`). Cero (0) se representa como `0000 0000` , y menos uno (-1) como `1111 1111` .

En general, sin embargo, esto significa `~n = -n - 1` .

```
# 0 = 0b0000 0000
~0
# Out: -1
# -1 = 0b1111 1111

# 1 = 0b0000 0001
~1
# Out: -2
# -2 = 1111 1110

# 2 = 0b0000 0010
~2
# Out: -3
# -3 = 0b1111 1101

# 123 = 0b0111 1011
~123
# Out: -124
# -124 = 0b1000 0100
```

Tenga en cuenta que el efecto general de esta operación cuando se aplica a números positivos se puede resumir:

```
~n -> -|n+1|
```

Y luego, cuando se aplica a números negativos, el efecto correspondiente es:

```
~-n -> |n-1|
```

Los siguientes ejemplos ilustran esta última regla ...

```
# -0 = 0b0000 0000
~-0
# Out: -1
# -1 = 0b1111 1111
# 0 is the obvious exception to this rule, as -0 == 0 always

# -1 = 0b1000 0001
~-1
# Out: 0
# 0 = 0b0000 0000

# -2 = 0b1111 1110
~-2
# Out: 1
# 1 = 0b0000 0001

# -123 = 0b1111 1011
~-123
# Out: 122
# 122 = 0b0111 1010
```

Operaciones in situ

Todos los operadores de Bitwise (excepto `~`) tienen sus propias versiones in situ

```
a = 0b001
a &= 0b010
# a = 0b000

a = 0b001
a |= 0b010
# a = 0b011

a = 0b001
a <<= 2
# a = 0b100

a = 0b100
a >>= 2
# a = 0b001

a = 0b101
a ^= 0b011
# a = 0b110
```

Lea Operadores de Bitwise en línea: https://riptutorial.com/es/python/topic/730/operadores-de-bitwise

Capítulo 149: Operadores matemáticos simples

Introducción

Python hace operadores matemáticos comunes por sí mismo, incluyendo división de números enteros y flotantes, multiplicación, exponenciación, suma y resta. El módulo matemático (incluido en todas las versiones estándar de Python) ofrece funciones ampliadas como funciones trigonométricas, operaciones de raíz, logaritmos y muchos más.

Observaciones

Tipos numéricos y sus metaclases.

El módulo de `numbers` contiene las metaclases abstractas para los tipos numéricos:

subclases	números.número	Números.Integral	números.Racional	numeros.Real	N
bool	✓	✓	✓	✓	✓
En t	✓	✓	✓	✓	✓
fracciones.Fracción	✓	-	✓	✓	✓
flotador	✓	-	-	✓	✓
complejo	✓	-	-	-	✓
decimal.decimal	✓	-	-	-	-

Examples

Adición

```
a, b = 1, 2

# Using the "+" operator:
a + b                   # = 3

# Using the "in-place" "+=" operator to add and assign:
a += b                  # a = 3 (equivalent to a = a + b)

import operator         # contains 2 argument arithmetic functions for the examples
```

```
operator.add(a, b)     # = 5  since a is set to 3 right before this line

# The "+=" operator is equivalent to:
a = operator.iadd(a, b)    # a = 5 since a is set to 3 right before this line
```

Posibles combinaciones (tipos incorporados):

- `int` e `int` (da un `int`)
- `int` y `float` (da un `float`)
- `int` y `complex` (da un `complex`).
- `float` y `float` (da un `float`)
- `float` y `complex` (da un `complex`).
- `complex` y `complex` (da un `complex`).

Nota: el operador `+` también se utiliza para concatenar cadenas, listas y tuplas:

```
"first string " + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

Sustracción

```
a, b = 1, 2

# Using the "-" operator:
b - a                  # = 1


import operator        # contains 2 argument arithmetic functions
operator.sub(b, a)     # = 1
```

Posibles combinaciones (tipos incorporados):

- `int` e `int` (da un `int`)
- `int` y `float` (da un `float`)
- `int` y `complex` (da un `complex`).
- `float` y `float` (da un `float`)
- `float` y `complex` (da un `complex`).
- `complex` y `complex` (da un `complex`).

Multiplicación

```
a, b = 2, 3

a * b                  # = 6

import operator
operator.mul(a, b)     # = 6
```

Posibles combinaciones (tipos incorporados):

- `int` e `int` (da un `int`)
- `int` y `float` (da un `float`)
- `int` y `complex` (da un `complex`).
- `float` y `float` (da un `float`)
- `float` y `complex` (da un `complex`).
- `complex` y `complex` (da un `complex`).

Nota: el operador `*` también se utiliza para la concatenación repetida de cadenas, listas y tuplas:

```
3 * 'ab'  # = 'abababab'
3 * ('a', 'b')  # = ('a', 'b', 'a', 'b', 'a', 'b')
```

División

Python hace división de enteros cuando ambos operandos son enteros. El comportamiento de los operadores de división de Python ha cambiado de Python 2.xy 3.x (ver también División de enteros).

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x 2.7

En Python 2, el resultado del operador '/' depende del tipo de numerador y denominador.

```
a / b                  # = 1

a / c                  # = 1.5

d / b                  # = -2

b / a                  # = 0

d / e                  # = -1
```

Tenga en cuenta que debido a que `a` y `b` son `int` s, el resultado es un `int` .

El resultado siempre se redondea hacia abajo (floored).

Debido a que `c` es un flotador, el resultado de `a / c` es un `float` .

También puede utilizar el módulo operador:

```
import operator        # the operator module provides 2-argument arithmetic functions
operator.div(a, b)     # = 1
operator.__div__(a, b) # = 1
```

Python 2.x 2.2

¿Qué tal si quieres división flotante?

Recomendado:

```
from __future__ import division # applies Python 3 style division to the entire module
a / b                   # = 1.5
a // b                  # = 1
```

De acuerdo (si no desea aplicar a todo el módulo):

```
a / (b * 1.0)           # = 1.5
1.0 * a / b             # = 1.5
a / b * 1.0             # = 1.0    (careful with order of operations)

from operator import truediv
truediv(a, b)           # = 1.5
```

No recomendado (puede generar TypeError, por ejemplo, si el argumento es complejo):

```
float(a) / b            # = 1.5
a / float(b)            # = 1.5
```

Python 2.x 2.2

El operador '//' en Python 2 fuerza la división de pisos independientemente del tipo.

```
a // b                # = 1
a // c                # = 1.0
```

Python 3.x 3.0

En Python 3, el operador `/` realiza una división "verdadera" independientemente de los tipos. El operador `//` realiza la división del piso y mantiene el tipo.

```
a / b                   # = 1.5
e / b                   # = 5.0
a // b                  # = 1
a // c                  # = 1.0

import operator             # the operator module provides 2-argument arithmetic functions
operator.truediv(a, b)      # = 1.5
operator.floordiv(a, b)     # = 1
operator.floordiv(a, c)     # = 1.0
```

Posibles combinaciones (tipos incorporados):

- `int` e `int` (da un `int` en Python 2 y un `float` en Python 3)
- `int` y `float` (da un `float`)
- `int` y `complex` (da un `complex`).
- `float` y `float` (da un `float`)
- `float` y `complex` (da un `complex`).

- `complex` y `complex` (da un `complex`).

Ver PEP 238 para más información.

Exponer

```
a, b = 2, 3

(a ** b)                # = 8
pow(a, b)               # = 8

import math
math.pow(a, b)          # = 8.0 (always float; does not allow complex results)

import operator
operator.pow(a, b)      # = 8
```

Otra diferencia entre el `pow` `math.pow` y `math.pow` es que el `pow` incorporado puede aceptar tres argumentos:

```
a, b, c = 2, 3, 2

pow(2, 3, 2)            # 0, calculates (2 ** 3) % 2, but as per Python docs,
                        #    does so more efficiently
```

Funciones especiales

La función `math.sqrt(x)` calcula la raíz cuadrada de `x` .

```
import math
import cmath
c = 4
math.sqrt(c)            # = 2.0 (always float; does not allow complex results)
cmath.sqrt(c)           # = (2+0j) (always complex)
```

Para calcular otras raíces, como una raíz cúbica, aumente el número al recíproco del grado de la raíz. Esto se podría hacer con cualquiera de las funciones exponenciales u operador.

```
 import math
 x = 8
 math.pow(x, 1/3) # evaluates to 2.0
 x**(1/3) # evaluates to 2.0
```

La función `math.exp(x)` calcula `e ** x` .

```
math.exp(0)  # 1.0
math.exp(1)  # 2.718281828459045 (e)
```

La función `math.expm1(x)` calcula `e ** x - 1` . Cuando `x` es pequeño, esto proporciona una precisión significativamente mejor que `math.exp(x) - 1` .

```
math.expm1(0)       # 0.0

math.exp(1e-6) - 1  # 1.0000004999621837e-06
math.expm1(1e-6)    # 1.0000005000001665e-06
# exact result      # 1.000000500000166666708333341666...
```

Logaritmos

De forma predeterminada, la función `math.log` calcula el logaritmo de un número, base e. Opcionalmente puede especificar una base como segundo argumento.

```
import math
import cmath

math.log(5)         # = 1.6094379124341003
# optional base argument. Default is math.e
math.log(5, math.e) # = 1.6094379124341003
cmath.log(5)        # = (1.6094379124341003+0j)
math.log(1000, 10)   # 3.0 (always returns float)
cmath.log(1000, 10)  # (3+0j)
```

Existen variaciones especiales de la función `math.log` para diferentes bases.

```
# Logarithm base e - 1 (higher precision for low values)
math.log1p(5)       # = 1.791759469228055

# Logarithm base 2
math.log2(8)        # = 3.0

# Logarithm base 10
math.log10(100)     # = 2.0
cmath.log10(100)    # = (2+0j)
```

Operaciones in situ

Es común que dentro de las aplicaciones se necesite tener un código como este:

```
a = a + 1
```

o

```
a = a * 2
```

Existe un atajo efectivo para estas operaciones in situ:

```
a += 1
# and
a *= 2
```

Se puede usar cualquier operador matemático antes del carácter '=' para realizar una operación in situ:

- `-=` disminuir la variable en su lugar
- `+=` incrementar la variable en su lugar
- `*=` multiplica la variable en su lugar
- `/=` dividir la variable en su lugar
- `//=` piso divide la variable en su lugar # Python 3
- `%=` devolver el módulo de la variable en su lugar
- `**=` elevar a una potencia en su lugar

Existen otros operadores in situ para los operadores bitwise (`^` , `|` etc)

Funciones trigonométricas

```
a, b = 1, 2

import math

math.sin(a)  # returns the sine of 'a' in radians
# Out: 0.8414709848078965

math.cosh(b)  # returns the inverse hyperbolic cosine of 'b' in radians
# Out: 3.7621956910836314

math.atan(math.pi)  # returns the arc tangent of 'pi' in radians
# Out: 1.2626272556789115

math.hypot(a, b) # returns the Euclidean norm, same as math.sqrt(a*a + b*b)
# Out: 2.23606797749979
```

> Tenga en cuenta que `math.hypot(x, y)` también es la longitud del vector (o distancia euclidiana) desde el origen `(0, 0)` hasta el punto `(x, y)` .
>
> Para calcular la distancia euclidiana entre dos puntos `(x1, y1)` y `(x2, y2)` puede usar `math.hypot` siguiente manera
>
> ```
> math.hypot(x2-x1, y2-y1)
> ```

Para convertir de radianes -> grados y grados -> radianes respectivamente use `math.degrees` y `math.radians`

```
math.degrees(a)
# Out: 57.29577951308232

math.radians(57.29577951308232)
# Out: 1.0
```

Módulo

Como en muchos otros idiomas, Python usa el operador `%` para calcular el módulo.

```
3 % 4     # 3
10 % 2    # 0
6 % 4     # 2
```

O utilizando el módulo `operator` :

```
import operator

operator.mod(3 , 4)      # 3
operator.mod(10 , 2)     # 0
operator.mod(6 , 4)      # 2
```

También puedes usar números negativos.

```
-9 % 7      # 5
9 % -7      # -5
-9 % -7     # -2
```

Si necesita encontrar el resultado de la división y el módulo de enteros, puede usar la función `divmod` como acceso directo:

```
quotient, remainder = divmod(9, 4)
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

Lea Operadores matemáticos simples en línea:
https://riptutorial.com/es/python/topic/298/operadores-matematicos-simples

Capítulo 150: Optimización del rendimiento

Observaciones

Cuando intente mejorar el rendimiento de un script de Python, en primer lugar debería poder encontrar el cuello de botella de su script y tener en cuenta que ninguna optimización puede compensar una mala elección en las estructuras de datos o una falla en el diseño de su algoritmo. La identificación de los cuellos de botella de rendimiento se puede hacer mediante la creación de perfiles de su script. En segundo lugar, no intente optimizar demasiado pronto su proceso de codificación a expensas de la legibilidad / diseño / calidad. Donald Knuth hizo la siguiente declaración sobre la optimización:

> "Debemos olvidarnos de las pequeñas eficiencias, digamos que aproximadamente el 97% de las veces: la optimización prematura es la raíz de todo mal. Sin embargo, no debemos dejar pasar nuestras oportunidades en ese 3% crítico ".

Examples

Código de perfil

En primer lugar, debe poder encontrar el cuello de botella de su script y tener en cuenta que ninguna optimización puede compensar una mala elección en la estructura de datos o una falla en el diseño de su algoritmo. En segundo lugar, no intente optimizar demasiado pronto su proceso de codificación a expensas de la legibilidad / diseño / calidad. Donald Knuth hizo la siguiente declaración sobre la optimización:

> "Debemos olvidarnos de las pequeñas eficiencias, digamos que aproximadamente el 97% de las veces: la optimización prematura es la raíz de todo mal. Sin embargo, no debemos dejar pasar nuestras oportunidades en ese 3% crítico".

Para perfilar su código tiene varias herramientas: `cProfile` (o el `profile` más lento) de la biblioteca estándar, `line_profiler` y `timeit` . Cada uno de ellos tiene un propósito diferente.

`cProfile` es un generador de perfiles determinístico: se controlan los eventos de llamada de función, retorno de función y excepción, y se realizan intervalos precisos para los intervalos entre estos eventos (hasta 0.001s). La documentación de la biblioteca ([https://docs.python.org/2/library/profile.html◆◆1]) nos proporciona un caso de uso simple

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

O si prefiere envolver partes de su código existente:

```
import cProfile, pstats, StringIO
```

```
pr = cProfile.Profile()
pr.enable()
# ... do something ...
# ... long ...
pr.disable()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

Esto creará resultados que se parecen a la tabla a continuación, donde puede ver rápidamente dónde pasa su programa la mayor parte de su tiempo e identificar las funciones para optimizar.

```
         3 function calls in 0.000 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(f)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

El módulo `line_profiler` ([https://github.com/rkern/line_profilerI [ ]] es útil para tener un análisis línea por línea de su código. Obviamente, esto no es manejable para scripts largos, pero está dirigido a fragmentos. Consulte la documentación para más detalles. La forma más fácil de comenzar es usar el script kernprof tal como se explica en la página del paquete, tenga en cuenta que deberá especificar manualmente la función (es) que desea perfilar.

```
$ kernprof -l script_to_profile.py
```

kernprof creará una instancia de LineProfiler y la insertará en el espacio de nombres `__builtins__` con el perfil de nombre. Se ha escrito para ser utilizado como decorador, por lo que en su script, decora las funciones que desea perfilar con `@profile` .

```
@profile
def slow_function(a, b, c):
    ...
```

El comportamiento predeterminado de kernprof es colocar los resultados en un archivo binario `script_to_profile.py.lprof` . Puede decirle a kernprof que vea inmediatamente los resultados formateados en el terminal con la opción [-v / - ver]. De lo contrario, puede ver los resultados más tarde así:

```
$ python -m line_profiler script_to_profile.py.lprof
```

Finalmente, `timeit` proporciona una forma sencilla de probar un forro o una pequeña expresión tanto desde la línea de comandos como desde el shell de python. Este módulo responderá preguntas como, ¿es más rápido hacer una lista de comprensión o usar la `list()` integrada `list()` al transformar un conjunto en una lista? Busque la palabra clave de `setup` o la opción `-s` para agregar el código de configuración.

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
```

desde una terminal

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
```

Lea Optimización del rendimiento en línea:
https://riptutorial.com/es/python/topic/5889/optimizacion-del-rendimiento

Capítulo 151: os.path

Introducción

Este módulo implementa algunas funciones útiles en las rutas de acceso. Los parámetros de ruta se pueden pasar como cadenas o bytes. Se recomienda que las aplicaciones representen los nombres de archivo como cadenas de caracteres (Unicode).

Sintaxis

- os.path.join (a, * p)
- os.path.basename (p)
- os.path.dirname (p)
- os.path.split (p)
- os.path.splitext (p)

Examples

Unir caminos

Para unir dos o más componentes de ruta, primero importe el módulo OS de Python y luego use lo siguiente:

```
import os
os.path.join('a', 'b', 'c')
```

La ventaja de usar os.path es que permite que el código permanezca compatible en todos los sistemas operativos, ya que utiliza el separador apropiado para la plataforma en la que se está ejecutando.

Por ejemplo, el resultado de este comando en Windows será:

```
>>> os.path.join('a', 'b', 'c')
'a\b\c'
```

En un sistema operativo Unix:

```
>>> os.path.join('a', 'b', 'c')
'a/b/c'
```

Camino Absoluto desde el Camino Relativo

Utilice `os.path.abspath` :

```
>>> os.getcwd()
'/Users/csaftoiu/tmp'
>>> os.path.abspath('foo')
'/Users/csaftoiu/tmp/foo'
>>> os.path.abspath('../foo')
'/Users/csaftoiu/foo'
>>> os.path.abspath('/foo')
'/foo'
```

Manipulación de componentes del camino

Para separar un componente de la ruta:

```
>>> p = os.path.join(os.getcwd(), 'foo.txt')
>>> p
'/Users/csaftoiu/tmp/foo.txt'
>>> os.path.dirname(p)
'/Users/csaftoiu/tmp'
>>> os.path.basename(p)
'foo.txt'
>>> os.path.split(os.getcwd())
('/Users/csaftoiu/tmp', 'foo.txt')
>>> os.path.splitext(os.path.basename(p))
('foo', '.txt')
```

Obtener el directorio padre

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

Si el camino dado existe.

para comprobar si existe el camino dado

```
path = '/home/john/temp'
os.path.exists(path)
#this returns false if path doesn't exist or if the path is a broken symbolic link
```

compruebe si la ruta dada es un directorio, archivo, enlace simbólico, punto de montaje, etc.

para comprobar si la ruta dada es un directorio

```
dirname = '/home/john/python'
os.path.isdir(dirname)
```

para comprobar si la ruta dada es un archivo

```
filename = dirname + 'main.py'
os.path.isfile(filename)
```

para comprobar si el camino dado es enlace simbólico

```
symlink = dirname + 'some_sym_link'
os.path.islink(symlink)
```

para comprobar si la ruta dada es un punto de montaje

```
mount_path = '/home'
os.path.ismount(mount_path)
```

Lea os.path en línea: https://riptutorial.com/es/python/topic/1380/os-path

Capítulo 152: Pandas Transform: Preforma operaciones en grupos y concatena los resultados.

Examples

Transformada simple

Primero, vamos a crear un marco de datos ficticio

Suponemos que un cliente puede tener n pedidos, un pedido puede tener m artículos y los artículos se pueden pedir más veces

```
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1,1,1,1,1,2,2,3,3,3,3,3]
orders_df['order_id'] = [1,1,1,2,2,3,3,4,5,6,6,6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry',
'strawberry']

# And this is how the dataframe looks like:
print(orders_df)
#     customer_id  order_id        item
# 0             1         1      apples
# 1             1         1   chocolate
# 2             1         1   chocolate
# 3             1         2      coffee
# 4             1         2      coffee
# 5             2         3      apples
# 6             2         3     bananas
# 7             3         4      coffee
# 8             3         5   milkshake
# 9             3         6   chocolate
# 10            3         6  strawberry
# 11            3         6  strawberry
```

.
.

Ahora, usaremos la función de `transform` pandas para contar el número de pedidos por cliente

```
# First, we define the function that will be applied per customer_id
count_number_of_orders = lambda x: len(x.unique())

# And now, we can tranform each group using the logic defined above
orders_df['number_of_orders_per_cient'] = (               # Put the results into a new column
```

```
that is called 'number_of_orders_per_cient'
                    orders_df                              # Take the original dataframe
                   .groupby(['customer_id'])['order_id'] # Create a seperate group for each
customer_id & select the order_id
                   .transform(count_number_of_orders))   # Apply the function to each group
seperatly

# Inspecting the results ...
print(orders_df)
#     customer_id  order_id        item  number_of_orders_per_cient
# 0             1         1      apples                           2
# 1             1         1   chocolate                           2
# 2             1         1   chocolate                           2
# 3             1         2      coffee                           2
# 4             1         2      coffee                           2
# 5             2         3      apples                           1
# 6             2         3     bananas                           1
# 7             3         4      coffee                           3
# 8             3         5   milkshake                           3
# 9             3         6   chocolate                           3
# 10            3         6  strawberry                           3
# 11            3         6  strawberry                           3
```

Múltiples resultados por grupo

Usando funciones de `transform` que devuelven sub-cálculos por grupo.

En el ejemplo anterior, tuvimos un resultado por cliente. Sin embargo, también se pueden aplicar funciones que devuelven valores diferentes para el grupo.

```
# Create a dummy dataframe
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1,1,1,1,1,2,2,3,3,3,3,3]
orders_df['order_id'] = [1,1,1,2,2,3,3,4,5,6,6,6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry',
'strawberry']


# Let's try to see if the items were ordered more than once in each orders

# First, we define a fuction that will be applied per group
def multiple_items_per_order(_items):
    # Apply .duplicated, which will return True is the item occurs more than once.
    multiple_item_bool = _items.duplicated(keep=False)
    return(multiple_item_bool)

# Then, we transform each group according to the defined function
orders_df['item_duplicated_per_order'] = (                    # Put the results into a new
column
                        orders_df                             # Take the orders dataframe
                        .groupby(['order_id'])['item']        # Create a seperate group for
each order_id & select the item
                        .transform(multiple_items_per_order)) # Apply the defined function to
```

```
each group separately

# Inspecting the results ...
print(orders_df)
#     customer_id  order_id        item  item_duplicated_per_order
# 0             1         1      apples                      False
# 1             1         1   chocolate                       True
# 2             1         1   chocolate                       True
# 3             1         2      coffee                       True
# 4             1         2      coffee                       True
# 5             2         3      apples                      False
# 6             2         3     bananas                      False
# 7             3         4      coffee                      False
# 8             3         5   milkshake                      False
# 9             3         6   chocolate                      False
# 10            3         6  strawberry                       True
# 11            3         6  strawberry                       True
```

Lea Pandas Transform: Preforma operaciones en grupos y concatena los resultados. en línea: https://riptutorial.com/es/python/topic/10947/pandas-transform--preforma-operaciones-en-grupos-y-concatena-los-resultados-

Capítulo 153: Patrones de diseño

Introducción

Un patrón de diseño es una solución general a un problema común en el desarrollo de software. Este tema de documentación está dirigido específicamente a proporcionar ejemplos de patrones de diseño comunes en Python.

Examples

Patrón de estrategia

Este patrón de diseño se llama patrón de estrategia. Se utiliza para definir una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. El patrón de diseño de estrategia permite que un algoritmo varíe independientemente de los clientes que lo utilizan.

Por ejemplo, los animales pueden "caminar" de muchas maneras diferentes. Caminar podría considerarse una estrategia que es implementada por diferentes tipos de animales:

```
from types import MethodType


class Animal(object):

    def __init__(self, *args, **kwargs):
        self.name = kwargs.pop('name', None) or 'Animal'
        if kwargs.get('walk', None):
            self.walk = MethodType(kwargs.pop('walk'), self)

    def walk(self):
        """
        Cause animal instance to walk

        Walking funcionallity is a strategy, and is intended to
        be implemented separately by different types of animals.
        """
        message = '{} should implement a walk method'.format(
            self.__class__.__name__)
        raise NotImplementedError(message)


# Here are some different walking algorithms that can be used with Animal
def snake_walk(self):
    print('I am slithering side to side because I am a {}.'.format(self.name))

def four_legged_animal_walk(self):
    print('I am using all four of my legs to walk because I am a(n) {}.'.format(
        self.name))

def two_legged_animal_walk(self):
    print('I am standing up on my two legs to walk because I am a {}.'.format(
        self.name))
```

Ejecutar este ejemplo produciría el siguiente resultado:

```
generic_animal = Animal()
king_cobra = Animal(name='King Cobra', walk=snake_walk)
elephant = Animal(name='Elephant', walk=four_legged_animal_walk)
kangaroo = Animal(name='Kangaroo', walk=two_legged_animal_walk)

kangaroo.walk()
elephant.walk()
king_cobra.walk()
# This one will Raise a NotImplementedError to let the programmer
# know that the walk method is intended to be used as a strategy.
generic_animal.walk()

    # OUTPUT:
    #
    # I am standing up on my two legs to walk because I am a Kangaroo.
    # I am using all four of my legs to walk because I am a(n) Elephant.
    # I am slithering side to side because I am a King Cobra.
    # Traceback (most recent call last):
    #   File "./strategy.py", line 56, in <module>
    #     generic_animal.walk()
    #   File "./strategy.py", line 30, in walk
    #     raise NotImplementedError(message)
    # NotImplementedError: Animal should implement a walk method
```

Tenga en cuenta que en lenguajes como C ++ o Java, este patrón se implementa utilizando una clase abstracta o una interfaz para definir una estrategia. En Python tiene más sentido simplemente definir algunas funciones externamente que pueden agregarse dinámicamente a una clase usando `types.MethodType` .

Introducción a los patrones de diseño y patrón Singleton.

Los patrones de diseño proporcionan soluciones a los `commonly occurring problems` en el diseño de software. Los patrones de diseño fueron introducidos por primera vez por `GoF(Gang of Four)` donde describían los patrones comunes como problemas que ocurren una y otra vez y soluciones a esos problemas.

Los patrones de diseño tienen cuatro elementos esenciales:

1. `The pattern name` es un identificador que podemos usar para describir un problema de diseño, sus soluciones y consecuencias en una o dos palabras.
2. `The problem` describe cuándo aplicar el patrón.
3. `The solution` describe los elementos que conforman el diseño, sus relaciones, responsabilidades y colaboraciones.
4. `The consequences` son los resultados y las compensaciones de aplicar el patrón.

Ventajas de los patrones de diseño:

1. Son reutilizables en múltiples proyectos.
2. El nivel arquitectónico de problemas puede ser resuelto.
3. Han sido probados y probados con el tiempo, lo cual es la experiencia de desarrolladores y arquitectos.

4. Tienen confiabilidad y dependencia.

Los patrones de diseño se pueden clasificar en tres categorías:

1. Patrón creacional
2. Patrón estructural
3. Patrón de comportamiento

`Creational Pattern` : se ocupan de cómo se puede crear el objeto y aíslan los detalles de la creación del objeto.

`Structural Pattern` : diseñan la estructura de clases y objetos para que puedan componerse para lograr resultados más grandes.

`Behavioral Pattern` : se ocupan de la interacción entre los objetos y la responsabilidad de los objetos.

Patrón Singleton :

Es un tipo de `creational pattern` que proporciona un mecanismo para tener solo uno y un objeto de un tipo dado y proporciona un punto de acceso global.

Por ejemplo, Singleton se puede usar en operaciones de base de datos, donde queremos que el objeto de la base de datos mantenga la consistencia de los datos.

Implementación

Podemos implementar Singleton Pattern en Python creando solo una instancia de la clase Singleton y sirviendo nuevamente el mismo objeto.

```
class Singleton(object):
    def __new__(cls):
        # hasattr method checks if the class object an instance property or not.
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

s = Singleton()
print ("Object created", s)

s1 = Singleton()
print ("Object2 created", s1)
```

Salida:

```
('Object created', <__main__.Singleton object at 0x10a7cc310>)
('Object2 created', <__main__.Singleton object at 0x10a7cc310>)
```

Tenga en cuenta que en lenguajes como C ++ o Java, este patrón se implementa haciendo que el constructor sea privado y creando un método estático que realice la inicialización del objeto. De esta manera, un objeto se crea en la primera llamada y la clase devuelve el mismo objeto a partir de entonces. Pero en Python, no tenemos forma de crear constructores privados.

Patrón de fábrica

El patrón de fábrica es también un `Creational pattern` . El término `factory` significa que una clase es responsable de crear objetos de otros tipos. Hay una clase que actúa como una fábrica que tiene objetos y métodos asociados con ella. El cliente crea un objeto llamando a los métodos con ciertos parámetros y la fábrica crea el objeto del tipo deseado y lo devuelve al cliente.

```
from abc import ABCMeta, abstractmethod

class Music():
    __metaclass__ = ABCMeta
    @abstractmethod
    def do_play(self):
        pass

class Mp3(Music):
    def do_play(self):
        print ("Playing .mp3 music!")

class Ogg(Music):
    def do_play(self):
        print ("Playing .ogg music!")

class MusicFactory(object):
    def play_sound(self, object_type):
        return eval(object_type)().do_play()

if __name__ == "__main__":
    mf = MusicFactory()
    music = input("Which music you want to play Mp3 or Ogg")
    mf.play_sound(music)
```

Salida:

```
Which music you want to play Mp3 or Ogg"Ogg"
Playing .ogg music!
```

`MusicFactory` es la clase de fábrica aquí que crea un objeto de tipo `Mp3` o `Ogg` según la elección que proporciona el usuario.

Apoderado

El objeto proxy se usa a menudo para garantizar el acceso protegido a otro objeto, cuya lógica empresarial interna no queremos contaminar con los requisitos de seguridad.

Supongamos que queremos garantizar que solo los usuarios con permisos específicos puedan acceder a los recursos.

Definición de proxy: (garantiza que solo los usuarios que realmente puedan ver las reservas puedan reservar el servicio al consumidor)

```
from datetime import date
from operator import attrgetter
```

```
class Proxy:
    def __init__(self, current_user, reservation_service):
        self.current_user = current_user
        self.reservation_service = reservation_service

    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        if self.current_user.can_see_reservations:
            return self.reservation_service.highest_total_price_reservations(
                date_from,
                date_to,
                reservations_count
              )
        else:
            return []

#Models and ReservationService:

class Reservation:
    def __init__(self, date, total_price):
        self.date = date
        self.total_price = total_price

class ReservationService:
    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        # normally it would be read from database/external service
        reservations = [
            Reservation(date(2014, 5, 15), 100),
            Reservation(date(2017, 5, 15), 10),
            Reservation(date(2017, 1, 15), 50)
        ]

        filtered_reservations = [r for r in reservations if (date_from <= r.date <= date_to)]

        sorted_reservations = sorted(filtered_reservations, key=attrgetter('total_price'),
reverse=True)

        return sorted_reservations[0:reservations_count]


class User:
    def __init__(self, can_see_reservations, name):
        self.can_see_reservations = can_see_reservations
        self.name = name

#Consumer service:

class StatsService:
    def __init__(self, reservation_service):
        self.reservation_service = reservation_service

    def year_top_100_reservations_average_total_price(self, year):
        reservations = self.reservation_service.highest_total_price_reservations(
            date(year, 1, 1),
            date(year, 12, 31),
            1
        )

        if len(reservations) > 0:
            total = sum(r.total_price for r in reservations)
```

```
            return total / len(reservations)
        else:
            return 0

#Test:
def test(user, year):
    reservations_service = Proxy(user, ReservationService())
    stats_service = StatsService(reservations_service)
    average_price = stats_service.year_top_100_reservations_average_total_price(year)
    print("{0} will see: {1}".format(user.name, average_price))

test(User(True, "John the Admin"), 2017)
test(User(False, "Guest"),         2017)
```

BENEFICIOS

- **estamos evitando cualquier cambio en `ReservationService` cuando se cambian las restricciones de acceso.**
- **no estamos mezclando datos relacionados con la empresa (`date_from` , `date_to` , `reservations_count`) con conceptos de dominio no relacionados (permisos de usuario) en servicio.**
- **El consumidor (`StatsService`) también está libre de la lógica relacionada con los permisos**

CUEVAS

- La interfaz de proxy es siempre exactamente la misma que el objeto que oculta, por lo que el usuario que consume el servicio envuelto por el proxy ni siquiera estaba al tanto de la presencia del proxy.

Lea Patrones de diseño en línea: https://riptutorial.com/es/python/topic/8056/patrones-de-diseno

Capítulo 154: Perfilado

Examples

%% timeit y% timeit en IPython

Concatanación de cadenas de perfiles:

```
In [1]: import string

In [2]: %%timeit s=""; long_list=list(string.ascii_letters)*50
  ....: for substring in long_list:
  ....:   s+=substring
  ....:
1000 loops, best of 3: 570 us per loop

In [3]: %%timeit long_list=list(string.ascii_letters)*50
  ....: s="".join(long_list)
  ....:
100000 loops, best of 3: 16.1 us per loop
```

Perfilado de bucles sobre iterables y listas:

```
In [4]: %timeit for i in range(100000):pass
100 loops, best of 3: 2.82 ms per loop

In [5]: %timeit for i in list(range(100000)):pass
100 loops, best of 3: 3.95 ms per loop
```

función timeit ()

Repetición de perfiles de elementos en una matriz.

```
>>> import timeit
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

línea de comandos de timeit

Perfil de concatanación de números.

```
python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 3: 29.2 usec per loop

python -m timeit "'-'.join(map(str,range(100)))"
100000 loops, best of 3: 19.4 usec per loop
```

line_profiler en linea de comando

El código fuente con la directiva @profile antes de la función que queremos perfilar:

```
import requests

@profile
def slow_func():
    s = requests.session()
    html=s.get("https://en.wikipedia.org/").text
    sum([pow(ord(x),3.1) for x in list(html)])

for i in range(50):
    slow_func()
```

Usando el comando kernprof para calcular el perfil línea por línea

```
$ kernprof -lv so6.py

Wrote profile results to so6.py.lprof
Timer unit: 4.27654e-07 s

Total time: 22.6427 s
File: so6.py
Function: slow_func at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def slow_func():
     6        50        20729    414.6      0.0      s = requests.session()
     7        50     47618627 952372.5     89.9
html=s.get("https://en.wikipedia.org/").text
     8        50      5306958 106139.2     10.0      sum([pow(ord(x),3.1) for x in
list(html)])
```

La solicitud de página es casi siempre más lenta que cualquier cálculo basado en la información de la página.

Usando cProfile (Perfilador preferido)

Python incluye un generador de perfiles llamado cProfile. Esto es generalmente preferido sobre el uso de timeit.

Desglosa su script completo y para cada método en su script le dice:

- `ncalls` : el número de veces que se llamó a un método
- `tottime` : tiempo total empleado en la función dada (excluyendo el tiempo realizado en llamadas a subfunciones)
- `percall` : Tiempo empleado por llamada. O el cociente de tottime dividido por ncalls
- `cumtime` : el tiempo acumulado empleado en esta y todas las subfunciones (desde la invocación hasta la salida). Esta cifra es precisa incluso para funciones recursivas.
- `percall` : es el cociente de cumtime dividido por llamadas primitivas

- `filename:lineno(function)` : proporciona los datos respectivos de cada función

El cProfiler se puede llamar fácilmente en la línea de comandos usando:

```
$ python -m cProfile main.py
```

Para ordenar la lista devuelta de métodos perfilados por el tiempo empleado en el método:

```
$ python -m cProfile -s time main.py
```

Lea Perfilado en línea: https://riptutorial.com/es/python/topic/3818/perfilado

Capítulo 155: Persistencia Python

Sintaxis

- pickle.dump (obj, file, protocol = None, *, fix_imports = True)
- pickle.load (archivo, *, fix_imports = True, encoding = "ASCII", errores = "strict")

Parámetros

Parámetro	Detalles
obj	Representación encurtida de obj al archivo de objeto de archivo abierto
protocolo	un entero, le dice al pickler que use el protocolo dado, `0` -ASCII, `1` - formato binario antiguo
expediente	El argumento del archivo debe tener un método write () `wb` para el método *dump* y para cargar el método `rb` read ()

Examples

Persistencia Python

Los objetos como números, listas, diccionarios, estructuras anidadas y objetos de instancia de clase viven en la memoria de su computadora y se pierden tan pronto como termina el script.

pickle almacena los datos de forma persistente en un archivo separado.

La representación de un objeto encurtido es siempre un objeto de bytes en todos los casos, por lo que uno debe abrir archivos en `wb` para almacenar datos y `rb` para cargar datos desde pickle.

Los datos pueden estar fuera de cualquier tipo, por ejemplo,

```
data={'a':'some_value',
     'b':[9,4,7],
     'c':['some_str','another_str','spam','ham'],
     'd':{'key':'nested_dictionary'},
     }
```

Almacenamiento de datos

```
import pickle
file=open('filename','wb')  #file object in binary write mode
pickle.dump(data,file)      #dump the data in the file object
file.close()                #close the file to write into the file
```

Cargar datos

```
import pickle
file=open('filename','rb')  #file object in binary read mode
data=pickle.load(file)      #load the data back
file.close()

>>>data
{'b': [9, 4, 7], 'a': 'some_value', 'd': {'key': 'nested_dictionary'},
 'c': ['some_str', 'another_str', 'spam', 'ham']}
```

Los siguientes tipos pueden ser decapados

1. Ninguno, verdadero y falso
2. enteros, números de punto flotante, números complejos
3. cuerdas, bytes, bytearrays
4. Tuplas, listas, conjuntos y diccionarios que contienen solo objetos extraíbles
5. Funciones definidas en el nivel superior de un módulo (usando def, no lambda)
6. Funciones incorporadas definidas en el nivel superior de un módulo
7. Clases que se definen en el nivel superior de un módulo.
8. instancias de dichas clases cuyo **dict** o el resultado de llamar a **getstate** ()

Función de utilidad para guardar y cargar.

Guardar datos en y desde archivo

```
import pickle
def save(filename,object):
    file=open(filename,'wb')
    pickle.dump(object,file)
    file.close()

def load(filename):
    file=open(filename,'rb')
    object=pickle.load(file)
    file.close()
    return object


>>>list_object=[1,1,2,3,5,8,'a','e','i','o','u']
>>>save(list_file,list_object)
>>>new_list=load(list_file)
>>>new_list
[1, 1, 2, 3, 5, 8, 'a', 'e', 'i', 'o', 'u'
```

Lea Persistencia Python en línea: https://riptutorial.com/es/python/topic/7810/persistencia-python

Capítulo 156: pip: PyPI Package Manager

Introducción

pip es el gestor de paquetes más utilizado para el Índice de paquetes de Python, instalado de forma predeterminada con las versiones recientes de Python.

Sintaxis

- pip <comando> [opciones] donde <comando> es uno de los siguientes:
  - instalar
    - Instalar paquetes
  - desinstalar
    - Desinstalar paquetes
  - congelar
    - Salida de paquetes instalados en formato de requerimientos.
  - lista
    - Listar paquetes instalados
  - espectáculo
    - Mostrar información sobre los paquetes instalados
  - buscar
    - Buscar PyPI para paquetes
  - rueda
    - Construye ruedas a partir de tus requerimientos.
  - cremallera
    - Zip paquetes individuales (obsoletos)
  - abrir la cremallera
    - Descomprimir paquetes individuales (obsoletos)
  - haz
    - Crear pybundles (en desuso)
  - ayuda
    - Mostrar ayuda para comandos

Observaciones

A veces, `pip` realizará una compilación manual de código nativo. En Linux, Python elegirá automáticamente un compilador de C disponible en su sistema. Consulte la tabla a continuación para obtener la versión requerida de Visual Studio / Visual C ++ en Windows (las versiones más recientes no funcionarán).

Versión Python	Versión de Visual Studio	Versión Visual C ++
2.6 - 3.2	Visual Studio 2008	Visual C ++ 9.0

Versión Python	Versión de Visual Studio	Versión Visual C ++
3.3 - 3.4	Visual Studio 2010	Visual C ++ 10.0
3.5	Visual Studio 2015	Visual C ++ 14.0

Fuente: wiki.python.org

Examples

Instalar paquetes

Para instalar la última versión de un paquete llamado `SomePackage` :

```
$ pip install SomePackage
```

Para instalar una versión específica de un paquete:

```
$ pip install SomePackage==1.0.4
```

Para especificar una versión mínima para instalar para un paquete:

```
$ pip install SomePackage>=1.0.4
```

Si los comandos muestran un error de denegación de permiso en Linux / Unix, use `sudo` con los comandos

---

Instalar desde archivos de requisitos

```
$ pip install -r requirements.txt
```

Cada línea del archivo de requisitos indica algo para instalar, y al igual que los argumentos para instalar, los detalles sobre el formato de los archivos están aquí: Formato del archivo de requisitos .

Después de instalar el paquete, puede verificarlo usando el comando de `freeze` :

```
$ pip freeze
```

Desinstalar paquetes

Para desinstalar un paquete:

```
$ pip uninstall SomePackage
```

Para listar todos los paquetes instalados usando `pip`

Para listar los paquetes instalados:

```
$ pip list
# example output
docutils (0.9.1)
Jinja2 (2.6)
Pygments (1.5)
Sphinx (1.1.2)
```

Para listar paquetes obsoletos y mostrar la última versión disponible:

```
$ pip list --outdated
# example output
docutils (Current: 0.9.1 Latest: 0.10)
Sphinx (Current: 1.1.2 Latest: 1.1.3)
```

Paquetes de actualización

Corriendo

```
$ pip install --upgrade SomePackage
```

actualizará el paquete `SomePackage` y todas sus dependencias. Además, pip elimina automáticamente la versión anterior del paquete antes de la actualización.

Para actualizar pip en sí, haz

```
$ pip install --upgrade pip
```

en Unix o

```
$ python -m pip install --upgrade pip
```

en las máquinas de Windows.

Actualizando todos los paquetes desactualizados en Linux

`pip` no contiene actualmente una bandera que permita a un usuario actualizar todos los paquetes desactualizados de una sola vez. Sin embargo, esto puede lograrse uniendo los comandos en un entorno Linux:

```
pip list --outdated --local | grep -v '^\-e' | cut -d = -f 1  | xargs -n1 pip install -U
```

Este comando toma todos los paquetes en el virtualenv local y comprueba si están desactualizados. De esa lista, obtiene el nombre del paquete y luego lo canaliza a un comando `pip install -U` . Al final de este proceso, todos los paquetes locales deben actualizarse.

Actualizando todos los paquetes desactualizados en Windows

`pip` no contiene actualmente una bandera que permita a un usuario actualizar todos los paquetes desactualizados de una sola vez. Sin embargo, esto puede lograrse uniendo los comandos en un entorno Windows:

```
for /F "delims= " %i in ('pip list --outdated --local') do pip install -U %i
```

Este comando toma todos los paquetes en el virtualenv local y comprueba si están desactualizados. De esa lista, obtiene el nombre del paquete y luego lo canaliza a un comando `pip install -U` . Al final de este proceso, todos los paquetes locales deben actualizarse.

Cree un archivo Requirements.txt de todos los paquetes en el sistema

`pip` ayuda en la creación de los `requirements.txt` archivos .txt al proporcionar la opción de `freeze` .

```
pip freeze > requirements.txt
```

Esto guardará una lista de todos los paquetes y su versión instalada en el sistema en un archivo llamado `requirements.txt` en la carpeta actual.

Cree un archivo Requirements.txt de paquetes solo en el virtualenv actual

`pip` ayuda en la creación de los `requirements.txt` archivos .txt al proporcionar la opción de `freeze` .

```
pip freeze --local > requirements.txt
```

El parámetro `--local` solo generará una lista de paquetes y versiones que se instalan localmente en un virtualenv. Los paquetes globales no serán listados.

Usando una determinada versión de Python con pip

Si tiene tanto Python 3 como Python 2 instalados, puede especificar qué versión de Python le gustaría que usara pip. Esto es útil cuando los paquetes solo admiten Python 2 o 3 o cuando desea probar con ambos.

Si desea instalar paquetes para Python 2, ejecute:

```
pip install [package]
```

o:

```
pip2 install [package]
```

Si desea instalar paquetes para Python 3, haga:

```
pip3 install [package]
```

También puede invocar la instalación de un paquete a una instalación específica de Python con:

```
\path\to\that\python.exe -m pip install some_package # on Windows OR
/usr/bin/python25 -m pip install some_package # on OS-X/Linux
```

En las plataformas OS-X / Linux / Unix, es importante tener en cuenta la distinción entre la versión del sistema de python (la actualización hace que el sistema deje de funcionar) y las versiones de usuario de python. Usted **puede** , *dependiendo de lo que está intentando actualizar* , necesita prefijar estos comandos con `sudo` e ingresar una contraseña.

Del mismo modo, en Windows, algunas instalaciones de Python, especialmente aquellas que son parte de otro paquete, pueden terminar instaladas en los directorios del sistema (aquellas que tendrá que actualizar desde una ventana de comandos que se ejecuta en el modo de administración) si le parece que necesita Para ello, es una **muy** buena idea comprobar qué instalación de Python está intentando actualizar con un comando como `python -c"import sys;print(sys.path);"` o `py -3.5 -c"import sys;print(sys.path);"` También puede comprobar qué pip está intentando ejecutar con `pip --version`

En Windows, si tienes ambos python 2 y python 3 instalados, y en tu ruta y tu python 3 es mayor que 3.4, entonces probablemente también tengas python launcher `py` en la ruta de tu sistema. A continuación, puede hacer trucos como:

```
py -3 -m pip install -U some_package # Install/Upgrade some_package to the latest python 3
py -3.3 -m pip install -U some_package # Install/Upgrade some_package to python 3.3 if present
py -2 -m pip install -U some_package # Install/Upgrade some_package to the latest python 2 -
64 bit if present
py -2.7-32 -m pip install -U some_package # Install/Upgrade some_package to python 2.7 - 32
bit if present
```

Si está ejecutando y manteniendo varias versiones de python, le recomendaría encarecidamente leer sobre los entornos `virtualenv` python `virtualenv` o `venv` que le permiten aislar tanto la versión de python como los paquetes que están presentes.

Instalación de paquetes aún no en pip como ruedas

Muchos paquetes de python puro aún no están disponibles en el Índice de Paquetes de Python como ruedas, pero aún así se instalan bien. Sin embargo, algunos paquetes en Windows le dan el error vcvarsall.bat no encontrado.

El problema es que el paquete que está intentando instalar contiene una extensión C o C ++ y no *está* disponible *actualmente* como una rueda *precompilada del* índice del paquete python, *pypi* , y en las ventanas no tiene la cadena de herramientas necesaria para compilar tales artículos

La respuesta más simple es ir al excelente sitio de Christoph Gohlke y localizar la versión **adecuada** de las bibliotecas que necesita. Por apropiado en el paquete nombrar **-cp *NN* -** tiene que coincidir con la versión de Python, es decir, si está utilizando Windows 32 bits pitón *incluso en Win64* el nombre debe incluir **-win32-** y si el uso de la pitón de 64 bits que debe incluir **-**

win_amd64 - y luego la versión de Python debe coincidir, es decir, para Python 34, el nombre del archivo **debe** incluir **-cp *34-*** , etc. Esta es básicamente la magia que el pip hace por ti en el sitio pypi.

Alternativamente, necesita obtener el kit de desarrollo de Windows apropiado para la versión de python que está utilizando, los encabezados de cualquier biblioteca en la que el paquete está intentando crear interfaces, posiblemente los encabezados de python para la versión de python, etc.

Python 2.7 usó Visual Studio 2008, Python 3.3 y 3.4 usó Visual Studio 2010, y Python 3.5+ usa Visual Studio 2015.

- Instale " Visual C ++ Compiler Package for Python 2.7 ", que está disponible en el sitio web de Microsoft **o**
- Instale " Windows SDK para Windows 7 y .NET Framework 4 " (v7.1), que está disponible en el sitio web de Microsoft **o**
- Instale Visual Studio 2015 Community Edition , *(o cualquier versión posterior, cuando se publiquen)* , **asegurándose de que selecciona las opciones para instalar el soporte de C & C ++ que** *ya no tiene el valor predeterminado . Me han dicho que la* descarga y la instalación *pueden demorar hasta* ***8 horas.*** así que **asegúrese de** que esas opciones estén configuradas en el primer intento.

Entonces es *posible que deba* ubicar los archivos de encabezado, *en la revisión* correspondiente de las bibliotecas a las que se vincula su paquete deseado y descargarlos en las ubicaciones correspondientes.

Finalmente , puede dejar que pip haga su compilación; por supuesto, si el paquete tiene dependencias que aún no tiene, es posible que también necesite encontrar los archivos de encabezado para ellos.

Alternativas: también vale la pena mirar hacia fuera, *tanto en pypi como en el sitio de Christop* , para cualquier versión ligeramente anterior del paquete que está buscando, ya sea python puro o pre-construido para su plataforma y versión de python y posiblemente usarlos, si encontrado, hasta que su paquete esté disponible. Del mismo modo, si está utilizando la última versión de python, es posible que los mantenedores de paquetes **necesiten** un poco de tiempo para ponerse al día, por lo que, para los proyectos que realmente **necesitan** un paquete específico, es posible que deba usar una python un poco más antigua por el momento. También puede consultar el sitio de origen de los paquetes para ver si hay una versión bifurcada que esté disponible preconstruida o como puro python y buscar paquetes alternativos que proporcionen la funcionalidad que necesita pero está disponible. Un ejemplo que viene a la mente es el siguiente. Almohada , *mantenida activamente* , reemplazo de PIL *actualmente no actualizado en 6 años y no disponible para python 3* .

Después , **recomendaría** a cualquier persona que tenga este problema que vaya al rastreador de errores del paquete y añada o aumente, si no hay uno ya, un boleto que solicite **educadamente** que los mantenedores del paquete proporcionen una rueda en pypi para su específico combinación de plataforma y python, si esto se hace, normalmente las cosas mejorarán con el tiempo, algunos mantenedores de paquetes no se dan cuenta de que se han perdido una

combinación determinada que las personas pueden estar usando.

Nota sobre la instalación de versiones preliminares

Pip sigue las reglas del control de versiones semántico y, de forma predeterminada, prefiere los paquetes publicados antes que los lanzamientos previos. Por lo tanto, si un paquete dado se ha liberado como `V0.98` y también hay una versión candidata `V1.0-rc1` el comportamiento predeterminado de `pip install` será instalar `V0.98` : si desea instalar la versión candidata, *le recomendamos para probar primero en un entorno virtual* , puede habilitarlo con `--pip install --pre` *package-name* o `--pip install --pre --upgrade` *package-name* . En muchos casos, los pre-lanzamientos o lanzamientos de candidatos pueden no tener ruedas creadas para todas las combinaciones de plataformas y versiones, por lo que es más probable que encuentre los problemas anteriores.

Nota sobre la instalación de versiones de desarrollo

También puede usar pip para instalar versiones de desarrollo de paquetes desde github y otras ubicaciones, ya que dicho código está en flujo, es muy poco probable que se le construyan ruedas, por lo que cualquier paquete impuro requerirá la presencia de las herramientas de construcción, y es posible que se puede romper en cualquier momento, por lo que se recomienda **encarecidamente** al usuario que solo instale dichos paquetes en un entorno virtual.

Existen tres opciones para tales instalaciones:

1. Descargue una instantánea comprimida, la mayoría de los sistemas de control de versiones en línea tienen la opción de descargar una instantánea comprimida del código. Esto puede descargarse manualmente y luego instalarse con la *ruta de* `pip install` */ to / download / file.* Tenga en cuenta que, para la mayoría de los formatos de compresión, pip manejará el desempaquetado en un área de caché, etc.
2. Permita que pip maneje la descarga e instale por usted con: `pip install` *URL / of / package / repository* - también puede necesitar usar las `--trusted-host` , `--client-cert` y / o `--proxy` para que esto funcione correctamente, especialmente en un entorno corporativo. p.ej:

```
    > py -3.5-32 -m venv demo-pip
    > demo-pip\Scripts\activate.bat
    > python -m pip install -U pip
    Collecting pip
      Using cached pip-9.0.1-py2.py3-none-any.whl
    Installing collected packages: pip
      Found existing installation: pip 8.1.1
        Uninstalling pip-8.1.1:
          Successfully uninstalled pip-8.1.1
    Successfully installed pip-9.0.1
    > pip install git+https://github.com/sphinx-doc/sphinx/
    Collecting git+https://github.com/sphinx-doc/sphinx/
      Cloning https://github.com/sphinx-doc/sphinx/ to c:\users\steve-
~1\appdata\local\temp\pip-04yn9hpp-build
    Collecting six>=1.5 (from Sphinx==1.7.dev20170506)
      Using cached six-1.10.0-py2.py3-none-any.whl
    Collecting Jinja2>=2.3 (from Sphinx==1.7.dev20170506)
```

```
      Using cached Jinja2-2.9.6-py2.py3-none-any.whl
    Collecting Pygments>=2.0 (from Sphinx==1.7.dev20170506)
      Using cached Pygments-2.2.0-py2.py3-none-any.whl
    Collecting docutils>=0.11 (from Sphinx==1.7.dev20170506)
      Using cached docutils-0.13.1-py3-none-any.whl
    Collecting snowballstemmer>=1.1 (from Sphinx==1.7.dev20170506)
      Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
    Collecting babel!=2.0,>=1.3 (from Sphinx==1.7.dev20170506)
      Using cached Babel-2.4.0-py2.py3-none-any.whl
    Collecting alabaster<0.8,>=0.7 (from Sphinx==1.7.dev20170506)
      Using cached alabaster-0.7.10-py2.py3-none-any.whl
    Collecting imagesize (from Sphinx==1.7.dev20170506)
      Using cached imagesize-0.7.1-py2.py3-none-any.whl
    Collecting requests>=2.0.0 (from Sphinx==1.7.dev20170506)
      Using cached requests-2.13.0-py2.py3-none-any.whl
    Collecting typing (from Sphinx==1.7.dev20170506)
      Using cached typing-3.6.1.tar.gz
    Requirement already satisfied: setuptools in f:\toolbuild\temp\demo-pip\lib\site-packages
(from Sphinx==1.7.dev20170506)
    Collecting sphinxcontrib-websupport (from Sphinx==1.7.dev20170506)
      Downloading sphinxcontrib_websupport-1.0.0-py2.py3-none-any.whl
    Collecting colorama>=0.3.5 (from Sphinx==1.7.dev20170506)
      Using cached colorama-0.3.9-py2.py3-none-any.whl
    Collecting MarkupSafe>=0.23 (from Jinja2>=2.3->Sphinx==1.7.dev20170506)
      Using cached MarkupSafe-1.0.tar.gz
    Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.7.dev20170506)
      Using cached pytz-2017.2-py2.py3-none-any.whl
    Collecting sqlalchemy>=0.9 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
      Downloading SQLAlchemy-1.1.9.tar.gz (5.2MB)
        100% |################################| 5.2MB 220kB/s
    Collecting whoosh>=2.0 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
      Downloading Whoosh-2.7.4-py2.py3-none-any.whl (468kB)
        100% |################################| 471kB 1.1MB/s
    Installing collected packages: six, MarkupSafe, Jinja2, Pygments, docutils,
snowballstemmer, pytz, babel, alabaster, imagesize, requests, typing, sqlalchemy, whoosh,
sphinxcontrib-websupport, colorama, Sphinx
      Running setup.py install for MarkupSafe ... done
      Running setup.py install for typing ... done
      Running setup.py install for sqlalchemy ... done
      Running setup.py install for Sphinx ... done
    Successfully installed Jinja2-2.9.6 MarkupSafe-1.0 Pygments-2.2.0 Sphinx-1.7.dev20170506
alabaster-0.7.10 babel-2.4.0 colorama-0.3.9 docutils-0.13.1 imagesize-0.7.1 pytz-2017.2
requests-2.13.0 six-1.10.0 snowballstemmer-1.2.1 sphinxcontrib-websupport-1.0.0 sqlalchemy-
1.1.9 typing-3.6.1 whoosh-2.7.4
```

Tenga en cuenta el prefijo `git+` a la URL.

3. Clonar el repositorio usando `git` , `mercurial` herramienta aceptable u otro, *preferentemente una herramienta DVCS,* y el uso `pip install` *ruta / a / clonada / repo* - esto **tanto** el proceso **como** cualquier archivo requires.text y llevar a cabo los pasos de generación y configuración, *se puede cambiar manualmente directorio a su repositorio clonado y ejecute* `pip install -r requires.txt` `python setup.py install` *y luego* `python setup.py install` *para obtener el mismo efecto.* Las grandes ventajas de este enfoque es que, si bien la operación de clonación inicial puede llevar más tiempo que la descarga de instantáneas, puede actualizar a la última versión con, en el caso de git: `git pull origin master` y si la versión actual contiene errores, puede usar la `pip uninstall` *nombre-paquete,* luego use los comandos de `git checkout` para retroceder a través del historial del repositorio a las

versiones anteriores y volver a intentarlo.

Lea pip: PyPI Package Manager en línea: https://riptutorial.com/es/python/topic/1781/pip--pypi-package-manager

Capítulo 157: Plantillas en python

Examples

Programa de salida de datos simple usando plantilla

```
from string import Template

data = dict(item = "candy", price = 8, qty = 2)

# define the template
t = Template("Simon bought $qty $item for $price dollar")
print(t.substitute(data))
```

Salida:

```
Simon bought 2 candy for 8 dollar
```

Las plantillas admiten sustituciones basadas en $ en lugar de sustituciones basadas en%. **Substitute** (mapeo, palabras clave) realiza la sustitución de plantillas, devolviendo una nueva cadena.

La asignación es cualquier objeto similar a un diccionario con claves que coinciden con los marcadores de posición de la plantilla. En este ejemplo, el precio y la cantidad son marcadores de posición. Los argumentos de palabras clave también se pueden utilizar como marcadores de posición. Los marcadores de posición de las palabras clave tienen prioridad si ambos están presentes.

Cambiando delimitador

Puede cambiar el delimitador "$" a cualquier otro. El siguiente ejemplo:

```
from string import Template

class MyOtherTemplate(Template):
    delimiter = "#"


data = dict(id = 1, name = "Ricardo")
t = MyOtherTemplate("My name is #name and I have the id: #id")
print(t.substitute(data))
```

Puedes leer los documentos aquí.

Lea Plantillas en python en línea: https://riptutorial.com/es/python/topic/6029/plantillas-en-python

Capítulo 158: Polimorfismo

Examples

Polimorfismo basico

El polimorfismo es la capacidad de realizar una acción en un objeto independientemente de su tipo. Esto generalmente se implementa creando una clase base y teniendo dos o más subclases que implementan todos los métodos con la misma firma. Cualquier otra función o método que manipule estos objetos puede llamar a los mismos métodos independientemente del tipo de objeto en el que esté operando, sin necesidad de realizar una verificación de tipo primero. En la terminología orientada a objetos, cuando la clase X extiende la clase Y, entonces Y se llama súper clase o clase base y X se llama subclase o clase derivada.

```
class Shape:
    """
    This is a parent class that is intended to be inherited by other classes
    """

    def calculate_area(self):
        """
        This method is intended to be overridden in subclasses.
        If a subclass doesn't implement it but it is called, NotImplemented will be raised.

        """
        raise NotImplemented

class Square(Shape):
    """
    This is a subclass of the Shape class, and represents a square
    """
    side_length = 2     # in this example, the sides are 2 units long

    def calculate_area(self):
        """
        This method overrides Shape.calculate_area(). When an object of type
        Square has its calculate_area() method called, this is the method that
        will be called, rather than the parent class' version.

        It performs the calculation necessary for this shape, a square, and
        returns the result.
        """
        return self.side_length * 2

class Triangle(Shape):
    """
    This is also a subclass of the Shape class, and it represents a triangle
    """
    base_length = 4
    height = 3

    def calculate_area(self):
        """
        This method also overrides Shape.calculate_area() and performs the area
```

```
        calculation for a triangle, returning the result.
        """

        return 0.5 * self.base_length * self.height

def get_area(input_obj):
    """
    This function accepts an input object, and will call that object's
    calculate_area() method. Note that the object type is not specified. It
    could be a Square, Triangle, or Shape object.
    """

    print(input_obj.calculate_area())

# Create one object of each class
shape_obj = Shape()
square_obj = Square()
triangle_obj = Triangle()

# Now pass each object, one at a time, to the get_area() function and see the
# result.
get_area(shape_obj)
get_area(square_obj)
get_area(triangle_obj)
```

Deberíamos ver esta salida:

Ninguna
4
6.0

¿Qué pasa sin el polimorfismo?
Sin polimorfismo, puede requerirse una verificación de tipo antes de realizar una acción en un objeto para determinar el método correcto para llamar. El siguiente **ejemplo de contador** realiza la misma tarea que el código anterior, pero sin el uso de polimorfismo, la función `get_area()` tiene que hacer más trabajo.

```
class Square:

    side_length = 2

    def calculate_square_area(self):
        return self.side_length ** 2

class Triangle:

    base_length = 4
    height = 3

    def calculate_triangle_area(self):
        return (0.5 * self.base_length) * self.height

def get_area(input_obj):

    # Notice the type checks that are now necessary here. These type checks
    # could get very complicated for a more complex example, resulting in
    # duplicate and difficult to maintain code.
```

```
    if type(input_obj).__name__ == "Square":
        area = input_obj.calculate_square_area()

    elif type(input_obj).__name__ == "Triangle":
        area = input_obj.calculate_triangle_area()

    print(area)

# Create one object of each class
square_obj = Square()
triangle_obj = Triangle()

# Now pass each object, one at a time, to the get_area() function and see the
# result.
get_area(square_obj)
get_area(triangle_obj)
```

Deberíamos ver esta salida:

```
4
6.0
```

Nota IMPORTANTE
Tenga en cuenta que las clases utilizadas en el ejemplo de contador son clases de "nuevo estilo" y se heredan implícitamente de la clase de objeto si se está utilizando Python 3. El polimorfismo funcionará tanto en Python 2.xy 3.x, pero el código de contraejemplo del polimorfismo generará una excepción si se ejecuta en un intérprete de Python 2.x porque escribe (input_obj). **el nombre** devolverá "instancia" en lugar del nombre de la clase si no se heredan explícitamente del objeto, lo que da como resultado que nunca se asigne un área.

Escribiendo pato

Polimorfismo sin herencia en forma de escritura de pato disponible en Python debido a su sistema de escritura dinámico. Esto significa que siempre que las clases contengan los mismos métodos, el intérprete de Python no distingue entre ellos, ya que la única comprobación de las llamadas se realiza en tiempo de ejecución.

```
class Duck:
    def quack(self):
        print("Quaaaaaack!")
    def feathers(self):
        print("The duck has white and gray feathers.")

class Person:
    def quack(self):
        print("The person imitates a duck.")
    def feathers(self):
        print("The person takes a feather from the ground and shows it.")
    def name(self):
        print("John Smith")

def in_the_forest(obj):
    obj.quack()
    obj.feathers()
```

```
donald = Duck()
john = Person()
in_the_forest(donald)
in_the_forest(john)
```

La salida es:

> Quaaaaaack!
> El pato tiene plumas blancas y grises.
> La persona imita a un pato.
> La persona toma una pluma del suelo y la muestra.

Lea Polimorfismo en línea: https://riptutorial.com/es/python/topic/5100/polimorfismo

Capítulo 159: PostgreSQL

Examples

Empezando

PostgreSQL es una base de datos de código abierto desarrollada activamente y madura. Usando el módulo `psycopg2` , podemos ejecutar consultas en la base de datos.

Instalación utilizando pip

```
pip install psycopg2
```

Uso básico

Supongamos que tenemos una tabla `my_table` en la base de datos `my_database` definida de la siguiente manera.

carné de identidad	nombre de pila	apellido
1	Juan	Gama

Podemos usar el módulo `psycopg2` para ejecutar consultas en la base de datos de la siguiente manera.

```
import psycopg2

# Establish a connection to the existing database 'my_database' using
# the user 'my_user' with password 'my_password'
con = psycopg2.connect("host=localhost dbname=my_database user=my_user password=my_password")

# Create a cursor
cur = con.cursor()

# Insert a record into 'my_table'
cur.execute("INSERT INTO my_table(id, first_name, last_name) VALUES (2, 'Jane', 'Doe');")

# Commit the current transaction
con.commit()

# Retrieve all records from 'my_table'
cur.execute("SELECT * FROM my_table;")
results = cur.fetchall()

# Close the database connection
con.close()

# Print the results
print(results)
```

```
# OUTPUT: [(1, 'John', 'Doe'), (2, 'Jane', 'Doe')]
```

Lea PostgreSQL en línea: https://riptutorial.com/es/python/topic/3374/postgresql

Capítulo 160: Precedencia del operador

Introducción

Los operadores de Python tienen un **orden de prioridad establecido** , que determina qué operadores se evalúan primero en una expresión potencialmente ambigua. Por ejemplo, en la expresión 3 * 2 + 7, primero se multiplica 3 por 2, y luego el resultado se agrega a 7, obteniendo 13. La expresión no se evalúa al revés, porque * tiene una precedencia más alta que +.

A continuación hay una lista de operadores por precedencia y una breve descripción de lo que hacen (generalmente).

Observaciones

De la documentación de Python:

> La siguiente tabla resume las precedencias del operador en Python, desde la precedencia más baja (menos vinculante) hasta la precedencia más alta (más vinculante). Los operadores en la misma caja tienen la misma prioridad. A menos que la sintaxis esté explícitamente dada, los operadores son binarios. Los operadores en el mismo grupo agrupan de izquierda a derecha (excepto las comparaciones, incluidas las pruebas, que tienen la misma prioridad y cadena de izquierda a derecha y exponenciación, que agrupa de derecha a izquierda).

Operador	Descripción
lambda	Expresión lambda
si	Expresión condicional
o	Booleano o
y	Booleano y
no x	Booleano no
en, no en, es, no es, <, <=,>,> =, <>,! =, ==	Comparaciones, incluyendo pruebas de membresía y pruebas de identidad
|	Bitwise o
^	Bitwise XOR
Y	Y a nivel de bit
<<, >>	Turnos

Operador	Descripción
+, -	Adición y sustracción
*, /, //,%	Multiplicación, división, resto [8]
+ x, -x, ~ x	Positivo, negativo, a nivel de bit NO
**	Exposiciónción [9]
x [índice], x [índice: índice], x (argumentos ...), x.attribute	Suscripción, corte, llamada, atributo de referencia.
(expresiones ...), [expresiones ...], {clave: valor ...}, expresiones ...	Encuadernación o visualización de tupla, visualización de lista, visualización de diccionario, conversión de cadena

Examples

Ejemplos simples de precedencia de operadores en python.

Python sigue la regla de PEMDAS. PEMDAS significa paréntesis, exponentes, multiplicación y división, y suma y resta.

Ejemplo:

```
>>> a, b, c, d = 2, 3, 5, 7
>>> a ** (b + c)  # parentheses
256
>>> a * b ** c  # exponent: same as `a * (b ** c)`
7776
>>> a + b * c / d  # multiplication / division: same as `a + (b * c / d)`
4.142857142857142
```

Extras: las reglas matemáticas se mantienen, pero no siempre :

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

Lea Precedencia del operador en línea: https://riptutorial.com/es/python/topic/5040/precedencia-del-operador

Capítulo 161: Procesos e hilos

Introducción

La mayoría de los programas se ejecutan línea por línea, ejecutando solo un proceso a la vez. Los hilos permiten que múltiples procesos fluyan independientemente unos de otros. El subprocesamiento con múltiples procesadores permite que los programas ejecuten múltiples procesos simultáneamente. Este tema documenta la implementación y el uso de subprocesos en Python.

Examples

Bloqueo de intérprete global

El rendimiento de subprocesos múltiples de Python a menudo puede verse afectado por el **bloqueo global de intérprete** . En resumen, aunque puede tener varios subprocesos en un programa de Python, solo una instrucción de bytecode puede ejecutarse en paralelo al mismo tiempo, independientemente del número de CPU.

Como tal, el multihilo en casos en los que las operaciones están bloqueadas por eventos externos, como el acceso a la red, puede ser bastante efectivo:

```
import threading
import time


def process():
    time.sleep(2)


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.00s
# Out: Four runs took 2.00s
```

Tenga en cuenta que aunque cada `process` tardó 2 segundos en ejecutarse, los cuatro procesos juntos pudieron ejecutarse de manera efectiva en paralelo, tomando un total de 2 segundos.

Sin embargo, los subprocesos múltiples en los casos en los que se realizan cálculos intensivos en

el código de Python, como muchos cálculos, no producen una gran mejora e incluso pueden ser más lentos que ejecutarse en paralelo:

```
import threading
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.05s
# Out: Four runs took 14.42s
```

En este último caso, el multiprocesamiento puede ser efectivo, ya que los procesos múltiples pueden, por supuesto, ejecutar múltiples instrucciones simultáneamente:

```
import multiprocessing
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))
```

```
start = time.time()
processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.07s
# Out: Four runs took 2.30s
```

Corriendo en múltiples hilos

Use `threading.Thread` para ejecutar una función en otro hilo.

```
import threading
import os

def process():
    print("Pid is %s, thread id is %s" % (os.getpid(), threading.current_thread().name))

threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()

# Out: Pid is 11240, thread id is Thread-1
# Out: Pid is 11240, thread id is Thread-2
# Out: Pid is 11240, thread id is Thread-3
# Out: Pid is 11240, thread id is Thread-4
```

Ejecutando en múltiples procesos

Use `multiprocessing.Process` para ejecutar una función en otro proceso. La interfaz es similar a `threading.Thread` :

```
import multiprocessing
import os

def process():
    print("Pid is %s" % (os.getpid(),))

processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()

# Out: Pid is 11206
# Out: Pid is 11207
# Out: Pid is 11208
# Out: Pid is 11209
```

Compartir el estado entre hilos

Como todos los subprocesos se ejecutan en el mismo proceso, todos los subprocesos tienen acceso a los mismos datos.

Sin embargo, el acceso simultáneo a los datos compartidos debe protegerse con un bloqueo para evitar problemas de sincronización.

```
import threading

obj = {}
obj_lock = threading.Lock()

def objify(key, val):
    print("Obj has %d values" % len(obj))
    with obj_lock:
        obj[key] = val
    print("Obj now has %d values" % len(obj))

ts = [threading.Thread(target=objify, args=(str(n), n)) for n in range(4)]
for t in ts:
    t.start()
for t in ts:
    t.join()
print("Obj final result:")
import pprint; pprint.pprint(obj)

# Out: Obj has 0 values
# Out:  Obj has 0 values
# Out: Obj now has 1 values
# Out: Obj now has 2 valuesObj has 2 values
# Out: Obj now has 3 values
# Out:
# Out:  Obj has 3 values
# Out: Obj now has 4 values
# Out: Obj final result:
# Out: {'0': 0, '1': 1, '2': 2, '3': 3}
```

Estado de intercambio entre procesos

El código que se ejecuta en diferentes procesos no comparte, de forma predeterminada, los mismos datos. Sin embargo, el módulo de `multiprocessing` contiene primitivas para ayudar a compartir valores a través de múltiples procesos.

```
import multiprocessing

plain_num = 0
shared_num = multiprocessing.Value('d', 0)
lock = multiprocessing.Lock()

def increment():
    global plain_num
    with lock:
        # ordinary variable modifications are not visible across processes
        plain_num += 1
        # multiprocessing.Value modifications are
```

```
        shared_num.value += 1

ps = [multiprocessing.Process(target=increment) for n in range(4)]
for p in ps:
    p.start()
for p in ps:
    p.join()

print("plain_num is %d, shared_num is %d" % (plain_num, shared_num.value))

# Out: plain_num is 0, shared_num is 4
```

Lea Procesos e hilos en línea: https://riptutorial.com/es/python/topic/4110/procesos-e-hilos

Capítulo 162: Programación Funcional en Python

Introducción

La programación funcional descompone un problema en un conjunto de funciones. Lo ideal es que las funciones solo tomen entradas y produzcan salidas, y no tengan ningún estado interno que afecte la salida producida para una entrada dada. A continuación, se encuentran las técnicas funcionales comunes a muchos idiomas: como lambda, map, reduce.

Examples

Función lambda

Una función anónima, en línea definida con lambda. Los parámetros de la lambda se definen a la izquierda de los dos puntos. El cuerpo de la función se define a la derecha de los dos puntos. El resultado de ejecutar el cuerpo de la función se devuelve (implícitamente).

```
s=lambda x:x*x
s(2)     =>4
```

Función de mapa

Mapa toma una función y una colección de elementos. Hace una nueva colección vacía, ejecuta la función en cada elemento de la colección original e inserta cada valor de retorno en la nueva colección. Devuelve la nueva colección.

Este es un mapa simple que toma una lista de nombres y devuelve una lista de las longitudes de esos nombres:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)     =>[4, 4, 3]
```

Función de reducción

Reducir toma una función y una colección de elementos. Devuelve un valor que se crea combinando los elementos.

Este es un simple reducir. Devuelve la suma de todos los elementos de la colección.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)     =>10
```

Función de filtro

Filtro toma una función y una colección. Devuelve una colección de cada elemento para el que la función devolvió True.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]    # outputs[5,6]
```

Lea Programación Funcional en Python en línea:
https://riptutorial.com/es/python/topic/9552/programacion-funcional-en-python

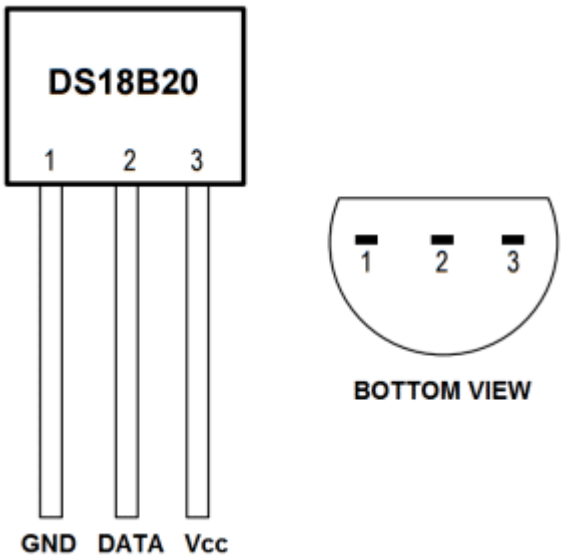
Capítulo 163: Programación IoT con Python y Raspberry PI

Examples

Ejemplo - sensor de temperatura

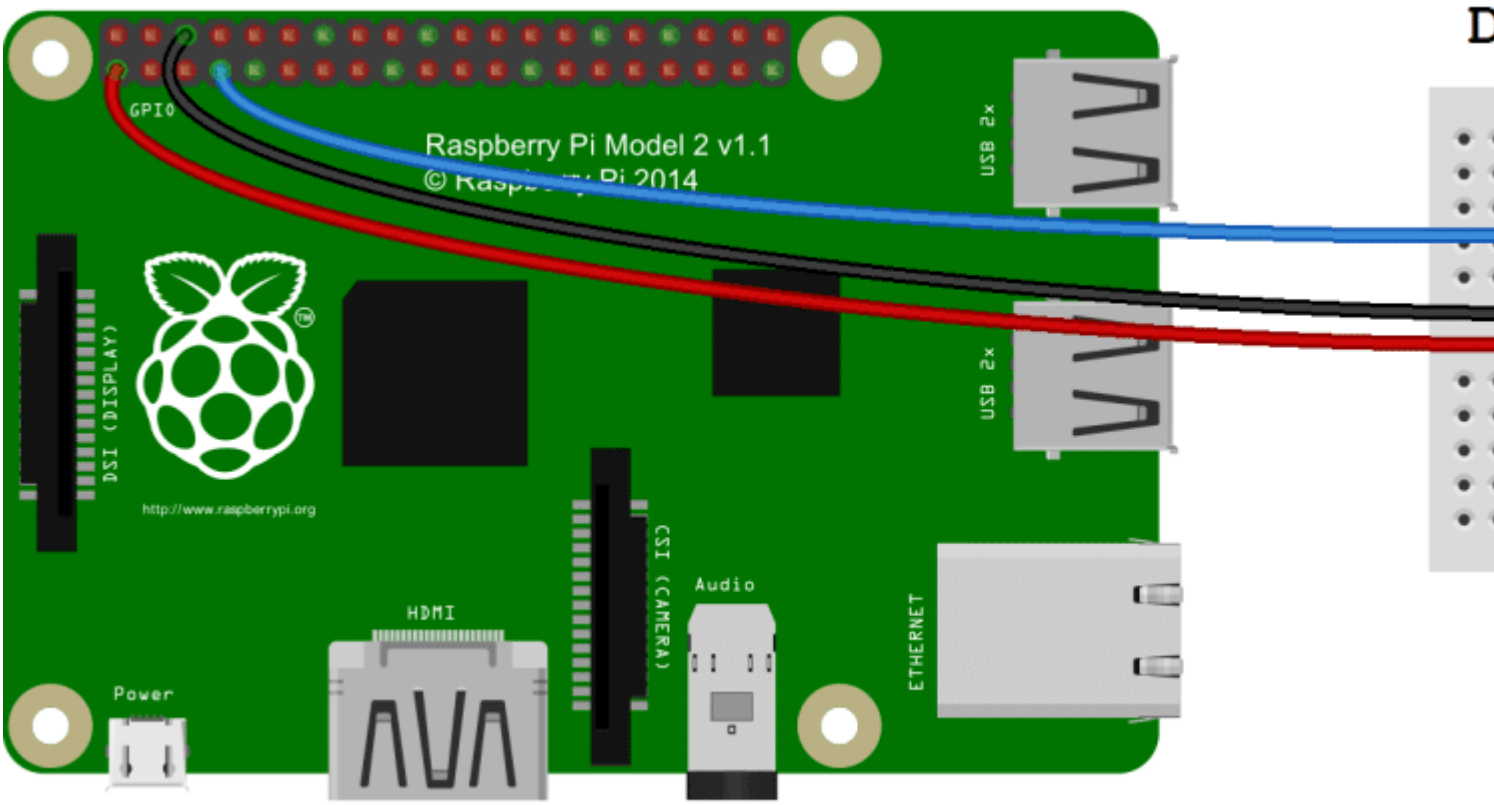
Interfaz de DS18B20 con Raspberry pi

Conexión de DS18B20 con Raspberry pi



Se puede ver que hay tres terminales.

1. Vcc
2. Gnd
3. Datos (protocolo de un cable)



R1 es una resistencia de 4.7k ohmios para elevar el nivel de voltaje

1. **Vcc** debe conectarse a cualquiera de los pines 5v o 3.3v de la Raspberry pi (PIN: 01, 02, 04, 17).
2. **Gnd** debe estar conectado a cualquiera de los pines Gnd de Raspberry pi (PIN: 06, 09, 14, 20, 25).
3. **Los datos** deben estar conectados a (PIN: 07)

Habilitando la interfaz de un cable desde el lado RPi

4. Inicie sesión en Raspberry pi utilizando putty o cualquier otro terminal linux / unix.
5. Después de iniciar sesión, abra el archivo /boot/config.txt en su navegador favorito.

   nano /boot/config.txt
6. Ahora agregue esta línea `dtoverlay=w1-gpio` al final del archivo.
7. Ahora reinicie el Raspberry pi `sudo reboot` .
8. Inicia sesión en Raspberry pi y ejecuta `sudo modprobe g1-gpio`
9. Luego ejecute `sudo modprobe w1-therm`
10. Ahora vaya al directorio / sys / bus / w1 / devices `cd /sys/bus/w1/devices`
11. Ahora descubrirá un directorio virtual creado con su sensor de temperatura a partir de 28 -

********.

12. Ir a este directorio `cd 28-********`

13. Ahora hay un nombre de archivo **w1-slave** , este archivo contiene la temperatura y otra información como CRC. `cat w1-slave` .

Ahora escribe un módulo en python para leer la temperatura

```
import glob
import time

RATE = 30
sensor_dirs = glob.glob("/sys/bus/w1/devices/28*")

if len(sensor_dirs) != 0:
    while True:
        time.sleep(RATE)
        for directories in sensor_dirs:
            temperature_file = open(directories + "/w1_slave")
            # Reading the files
            text = temperature_file.read()
            temperature_file.close()
            # Split the text with new lines (\n) and select the second line.
            second_line = text.split("\n")[1]
            # Split the line into words, and select the 10th word
            temperature_data = second_line.split(" ")[9]
            # We will read after ignoring first two character.
            temperature = float(temperature_data[2:])
            # Now normalise the temperature by dividing 1000.
            temperature = temperature / 1000
            print 'Address : '+str(directories.split('/')[-1])+', Temperature :
'+str(temperature)
```

Sobre el módulo de python se imprimirá la temperatura frente a la dirección durante un tiempo infinito. El parámetro RATE se define para cambiar o ajustar la frecuencia de la consulta de temperatura del sensor.

Diagrama de pin GPIO

1. [https://www.element14.com/community/servlet/JiveServlet/previewBody/73950-102-11-339300/pi3_gpio.png◆[3]

Lea Programación IoT con Python y Raspberry PI en línea:
https://riptutorial.com/es/python/topic/10735/programacion-iot-con-python-y-raspberry-pi

Capítulo 164: py.test

Examples

Configurando py.test

`py.test` es una de varias bibliotecas de pruebas de terceros que están disponibles para Python. Se puede instalar utilizando `pip` con

```
pip install pytest
```

El código a probar

Digamos que estamos probando una función de adición en `projectroot/module/code.py` :

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

El código de prueba

Creamos un archivo de prueba en `projectroot/tests/test_code.py` . El archivo **debe comenzar con `test_` ** para que se reconozca como un archivo de prueba.

```
# projectroot/tests/test_code.py
from module import code


def test_add():
    assert code.add(1, 2) == 3
```

Corriendo la prueba

Desde `projectroot` simplemente ejecutamos `py.test` :

```
# ensure we have the modules
$ touch tests/__init__.py
$ touch module/__init__.py
$ py.test
============================================== test session starts
=============================================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py .
```

```
================================================ 1 passed in 0.01 seconds
================================================
```

Pruebas de falla

Una prueba fallida proporcionará resultados útiles en cuanto a lo que salió mal:

```
# projectroot/tests/test_code.py
from module import code


def test_add__failing():
    assert code.add(10, 11) == 33
```

Resultados:

```
$ py.test
================================================= test session starts
=================================================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py F

====================================================== FAILURES
======================================================
_____________________________________________________ test_add__failing
_____________________________________________________

    def test_add__failing():
>       assert code.add(10, 11) == 33
E       assert 21 == 33
E        +  where 21 = <function add at 0x105d4d6e0>(10, 11)
E        +    where <function add at 0x105d4d6e0> = code.add

tests/test_code.py:5: AssertionError
================================================ 1 failed in 0.01 seconds
================================================
```

Introducción a los accesorios de prueba

Las pruebas más complicadas a veces necesitan tener las cosas configuradas antes de ejecutar el código que desea probar. Es posible hacer esto en la función de prueba en sí, pero luego terminas con funciones de prueba grandes que hacen tanto que es difícil decir dónde se detiene la configuración y comienza la prueba. También puede obtener una gran cantidad de códigos de configuración duplicados entre sus diversas funciones de prueba.

Nuestro archivo de código:

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
```

```
        self.bar = 2
```

Nuestro archivo de prueba:

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def test_foo_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Estos son ejemplos bastante simples, pero si nuestro objeto `Stuff` necesitara mucha más configuración, se volvería difícil de manejar. Vemos que hay un código duplicado entre nuestros casos de prueba, así que vamos a reformular eso en una función separada primero.

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def get_prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff


def test_foo_updates():
    my_stuff = get_prepped_stuff()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = get_prepped_stuff()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Esto se ve mejor, pero todavía tenemos la `my_stuff = get_prepped_stuff()` nuestras funciones de prueba.

Py.test accesorios para el rescate!

Los accesorios son versiones mucho más potentes y flexibles de las funciones de configuración de prueba. Pueden hacer mucho más de lo que estamos aprovechando aquí, pero lo haremos paso a paso.

Primero cambiamos `get_prepped_stuff` a un dispositivo llamado `prepped_stuff` . Desea nombrar sus aparatos con sustantivos en lugar de verbos debido a la forma en que los aparatos se utilizarán en las funciones de prueba más adelante. El `@pytest.fixture` indica que esta función específica debe manejarse como un accesorio en lugar de una función regular.

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

Ahora debemos actualizar las funciones de prueba para que usen el aparato. Esto se hace agregando un parámetro a su definición que coincida exactamente con el nombre del dispositivo. Cuando se ejecuta py.test, ejecutará el dispositivo antes de ejecutar la prueba, luego pasará el valor de retorno del dispositivo a la función de prueba a través de ese parámetro. (Tenga en cuenta que los dispositivos no **necesitan** devolver un valor; en su lugar, pueden hacer otras tareas de configuración, como llamar a un recurso externo, organizar las cosas en el sistema de archivos, poner valores en una base de datos, independientemente de las pruebas necesarias para la configuración)

```
def test_foo_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Ahora puedes ver por qué lo nombramos con un sustantivo. pero la línea `my_stuff = prepped_stuff` es bastante inútil, así que simplemente utilicemos `prepped_stuff` directamente.

```
def test_foo_updates(prepped_stuff):
    assert 1 == prepped_stuff.foo
    prepped_stuff.foo = 30000
    assert prepped_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    assert 2 == prepped_stuff.bar
    prepped_stuff.bar = 42
    assert 42 == prepped_stuff.bar
```

Ahora estamos usando accesorios! Podemos ir más lejos cambiando el alcance del dispositivo

(de modo que solo se ejecute una vez por módulo de prueba o sesión de ejecución del conjunto de pruebas en lugar de una vez por función de prueba), construyendo dispositivos que utilicen otros dispositivos, parametrizando el dispositivo (para que el dispositivo y todo las pruebas que usan ese dispositivo se ejecutan varias veces, una vez para cada parámetro dado al dispositivo), dispositivos que leen los valores del módulo que los llama ... como se mencionó anteriormente, los dispositivos tienen mucha más potencia y flexibilidad que una función de configuración normal.

Limpieza después de las pruebas.

Digamos que nuestro código ha crecido y nuestro objeto Stuff ahora necesita una limpieza especial.

```
# projectroot/module/stuff.py
class Stuff(object):
def prep(self):
    self.foo = 1
    self.bar = 2

def finish(self):
    self.foo = 0
    self.bar = 0
```

Podríamos agregar algún código para llamar a la limpieza al final de cada función de prueba, pero los dispositivos proporcionan una mejor manera de hacerlo. Si agrega una función al dispositivo y lo registra como **finalizador** , se llamará al código en la función de finalizador después de que se realice la prueba con el dispositivo. Si el alcance del dispositivo es mayor que una sola función (como módulo o sesión), el finalizador se ejecutará una vez que se hayan completado todas las pruebas en el alcance, por lo que una vez que el módulo haya terminado de ejecutarse o al final de toda la sesión de ejecución de prueba .

```
@pytest.fixture
def prepped_stuff(request):  # we need to pass in the request to use finalizers
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    def fin():  # finalizer function
        # do all the cleanup here
        my_stuff.finish()
    request.addfinalizer(fin)  # register fin() as a finalizer
    # you can do more setup here if you really want to
    return my_stuff
```

Usar la función de finalizador dentro de una función puede ser un poco difícil de entender a primera vista, especialmente cuando tienes dispositivos más complicados. En su lugar, puede utilizar un **dispositivo de rendimiento** para hacer lo mismo con un flujo de ejecución legible más humano. La única diferencia real es que, en lugar de utilizar la `return` , usamos un `yield` en la parte del dispositivo donde se realiza la configuración y el control debe ir a una función de prueba, luego se agrega todo el código de limpieza después del `yield` . También lo decoramos como un `yield_fixture` para que py.test sepa cómo manejarlo.

```
@pytest.yield_fixture
```

```
def prepped_stuff():  # it doesn't need request now!
    # do setup
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    # setup is done, pass control to the test functions
    yield my_stuff
    # do cleanup
    my_stuff.finish()
```

¡Y eso concluye la Introducción a los Aparatos de Prueba!

Para obtener más información, consulte la documentación oficial de la prueba py.test y la documentación oficial de la unidad de rendimiento.

Lea py.test en línea: https://riptutorial.com/es/python/topic/7054/py-test

Capítulo 165: pyaudio

Introducción

PyAudio proporciona enlaces de Python para PortAudio, la biblioteca de E / S de audio multiplataforma. Con PyAudio, puede usar Python fácilmente para reproducir y grabar audio en una variedad de plataformas. PyAudio está inspirado en:

1.pyPortAudio / fastaudio: enlaces de Python para la API de PortAudio v18.

2.tkSnack: kit de herramientas de sonido multiplataforma para Tcl / Tk y Python.

Observaciones

Nota: se llama a stream_callback en un hilo separado (del hilo principal). Las excepciones que se producen en el stream_callback:
1. Imprima un rastreo de error estándar para ayudar a la depuración,
2 .queue la excepción que se lanzará (en algún momento) en el hilo principal, y
3. Devuelva paAbort a PortAudio para detener la transmisión.
Nota: No llame a Stream.read () ni a Stream.write () si usa una operación sin bloqueo.
Ver: firma de devolución de llamada de PortAudio para detalles adicionales:
http://portaudio.com/docs/v19-doxydocs/portaudio_8h.html#a8a60fb2a5ec9cbade3f54a9c978e2710

Examples

Modo de devolución de llamada de audio I / O

```
"""PyAudio Example: Play a wave file (callback version)."""

import pyaudio
import wave
import time
import sys

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
p = pyaudio.PyAudio()

# define callback (2)
def callback(in_data, frame_count, time_info, status):
    data = wf.readframes(frame_count)
    return (data, pyaudio.paContinue)
```

```
# open stream using callback (3)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True,
                stream_callback=callback)

# start the stream (4)
stream.start_stream()

# wait for stream to finish (5)
while stream.is_active():
    time.sleep(0.1)

# stop stream (6)
stream.stop_stream()
stream.close()
wf.close()

# close PyAudio (7)
p.terminate()
```

En el modo de devolución de llamada, PyAudio llamará a una función de devolución de llamada especificada (2) siempre que necesite nuevos datos de audio (para reproducir) y / o cuando haya nuevos datos de audio (grabados) disponibles. Tenga en cuenta que PyAudio llama a la función de devolución de llamada en un hilo separado. La función tiene la siguiente `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` firma `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` y debe devolver una tupla que contiene fotogramas `frame_count` de datos de audio y una bandera que indica si hay más frames para reproducir / grabar.

Comience a procesar la transmisión de audio utilizando **pyaudio.Stream.start_stream ()** (4), que llamará repetidamente a la función de devolución de llamada hasta que esa función devuelva **pyaudio.paComplete** .

Para mantener el flujo activo, el hilo principal no debe terminar, por ejemplo, durmiendo (5).

Modo de bloqueo de E / S de audio

"" "Ejemplo de PyAudio: reproducir un archivo wave." ""

```
import pyaudio
import wave
import sys

CHUNK = 1024

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
p = pyaudio.PyAudio()
```

```
# open stream (2)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True)

# read data
data = wf.readframes(CHUNK)

# play stream (3)
while len(data) > 0:
    stream.write(data)
    data = wf.readframes(CHUNK)

# stop stream (4)
stream.stop_stream()
stream.close()

# close PyAudio (5)
p.terminate()
```

Para usar PyAudio, primero cree una instancia de PyAudio usando **pyaudio.PyAudio ()** (1), que configura el sistema portaudio.

Para grabar o reproducir audio, abra una transmisión en el dispositivo deseado con los parámetros de audio deseados utilizando **pyaudio.PyAudio.open ()** (2). Esto configura un **pyaudio.Stream** para reproducir o grabar audio.

Reproduce el audio escribiendo datos de audio en la transmisión usando **pyaudio.Stream.write ()** , o lee los datos de audio de la transmisión usando **pyaudio.Stream.read ()** . (3)

Tenga en cuenta que en el " *modo de bloqueo* ", cada **pyaudio.Stream.write ()** o **pyaudio.Stream.read () se** bloquea hasta que todos los cuadros dados / solicitados se hayan reproducido / grabado. Alternativamente, para generar datos de audio sobre la marcha o procesar inmediatamente los datos de audio grabados, use el "modo de devolución de llamada" (*consulte el ejemplo en el modo de devolución de llamada*)

Use pyaudio.Stream.stop_stream () para pausar la reproducción / grabación, y **pyaudio.Stream.close ()** para terminar la transmisión. (4)

Finalmente, finalice la sesión de **portaudio** utilizando **pyaudio.PyAudio.terminate ()** (5)

Lea pyaudio en línea: https://riptutorial.com/es/python/topic/10627/pyaudio

Capítulo 166: pygame

Introducción

Pygame es la biblioteca de acceso para hacer aplicaciones multimedia, especialmente juegos, en Python. El sitio web oficial es http://www.pygame.org/ .

Sintaxis

- pygame.mixer.init (frecuencia = 22050, tamaño = -16, canales = 2, búfer = 4096)
- pygame.mixer.pre_init (frecuencia, tamaño, canales, búfer)
- pygame.mixer.quit ()
- pygame.mixer.get_init ()
- pygame.mixer.stop ()
- pygame.mixer.pause ()
- pygame.mixer.unpause ()
- pygame.mixer.fadeout (tiempo)
- pygame.mixer.set_num_channels (count)
- pygame.mixer.get_num_channels ()
- pygame.mixer.set_reserved (cuenta)
- pygame.mixer.find_channel (force)
- pygame.mixer.get_busy ()

Parámetros

Parámetro	Detalles
contar	Un entero positivo que representa algo así como la cantidad de canales necesarios para ser reservados.
fuerza	Un valor booleano (`False` o `True`) que determina si `find_channel()` tiene que devolver un canal (inactivo o no) con `True` o no (si no hay canales inactivos) con `False`

Examples

Instalando pygame

Con `pip`

```
pip install pygame
```

Con `conda` :

```
conda install -c tlatorre pygame=1.9.2
```

Descarga directa desde el sitio web: http://www.pygame.org/download.shtml

Puede encontrar los instaladores adecuados para Windows y otros sistemas operativos.

Los proyectos también se pueden encontrar en http://www.pygame.org/

Modulo mezclador de pygame

El módulo `pygame.mixer` ayuda a controlar la música utilizada en los programas de `pygame` . A partir de ahora, hay 15 funciones diferentes para el módulo `mixer` .

Inicializando

De manera similar a cómo tienes que inicializar `pygame` con `pygame.init()` , también debes inicializar `pygame.mixer` .

Al usar la primera opción, inicializamos el módulo usando los valores predeterminados. Puede, sin embargo, anular estas opciones predeterminadas. Al usar la segunda opción, podemos inicializar el módulo usando los valores que manualmente ingresamos en nosotros mismos. Valores estándar:

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

Para verificar si lo hemos inicializado o no, podemos usar `pygame.mixer.get_init()` , que devuelve `True` si es y `False` si no lo es. Para salir / deshacer la inicialización, simplemente use `pygame.mixer.quit()` . Si desea continuar reproduciendo sonidos con el módulo, es posible que deba reinicializar el módulo.

Posibles acciones

Mientras se reproduce su sonido, puede pausarlo temporalmente con `pygame.mixer.pause()` . Para reanudar la reproducción de sus sonidos, simplemente use `pygame.mixer.unpause()` . También puedes desvanecer el final del sonido utilizando `pygame.mixer.fadeout()` . Se necesita un argumento, que es la cantidad de milisegundos que se tarda en terminar de desvanecerse la música.

Los canales

Puede reproducir tantas canciones como sea necesario siempre que haya suficientes canales abiertos para admitirlas. Por defecto, hay 8 canales. Para cambiar la cantidad de canales que hay, use `pygame.mixer.set_num_channels()` . El argumento es un entero no negativo. Si la cantidad de canales disminuye, cualquier sonido que se reproduzca en los canales eliminados se detendrá

de inmediato.

Para saber cuántos canales se están utilizando actualmente, llame a `pygame.mixer.get_channels(count)` . La salida es el número de canales que no están abiertos actualmente. También puede reservar canales para los sonidos que deben reproducirse utilizando `pygame.mixer.set_reserved(count)` . El argumento también es un entero no negativo. Cualquier sonido que se reproduzca en los canales recién reservados no se detendrá.

También puede averiguar qué canal no se está utilizando mediante el uso de `pygame.mixer.find_channel(force)` . Su argumento es un bool: verdadero o falso. Si no hay canales que estén inactivos y la `force` sea Falso, devolverá `None` . Si la `force` es verdadera, devolverá el canal que ha estado tocando durante más tiempo.

Lea pygame en línea: https://riptutorial.com/es/python/topic/8761/pygame

Capítulo 167: Pyglet

Introducción

Pyglet es un módulo de Python utilizado para efectos visuales y de sonido. No tiene dependencias en otros módulos. Ver [pyglet.org] [1] para la información oficial. [1]: http://pyglet.org

Examples

Hola Mundo en Pyglet

```
import pyglet
window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
                      font_name='Times New Roman',
                      font_size=36,
                      x=window.width//2, y=window.height//2,
                      anchor_x='center', anchor_y='center')
@window.event
def on_draw():
    window.clear()
    label.draw()
pyglet.app.run()
```

Instalación de Pyglet

Instala Python, entra en la línea de comandos y escribe:

Python 2:

```
pip install pyglet
```

Python 3:

```
pip3 install pyglet
```

Reproducción de sonido en Pyglet

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

Usando Pyglet para OpenGL

```
import pyglet
from pyglet.gl import *
```

```
win = pyglet.window.Window()

@win.event()
def on_draw():
    #OpenGL goes here. Use OpenGL as normal.

pyglet.app.run()
```

Dibujar puntos usando Pyglet y OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()
glClear(GL_COLOR_BUFFER_BIT)

@win.event
def on_draw():
    glBegin(GL_POINTS)
    glVertex2f(x, y) #x is desired distance from left side of window, y is desired distance
from bottom of window
    #make as many vertexes as you want
    glEnd
```

Para conectar los puntos, reemplace `GL_POINTS` con `GL_LINE_LOOP` .

Lea Pyglet en línea: https://riptutorial.com/es/python/topic/8208/pyglet

Capítulo 168: PyInstaller - Distribuir código de Python

Sintaxis

- pyinstaller [opciones] script [script ...] | archivo de especificaciones

Observaciones

PyInstaller es un módulo utilizado para agrupar aplicaciones de Python en un solo paquete junto con todas las dependencias. El usuario puede ejecutar la aplicación del paquete sin un intérprete de Python o cualquier módulo. Combina correctamente muchos paquetes importantes como numpy, Django, OpenCv y otros.

Algunos puntos importantes para recordar:

- Pyinstaller es compatible con Python 2.7 y Python 3.3+
- Pyinstaller ha sido probado contra Windows, Linux y Mac OS X.
- **NO** es un compilador cruzado. (Una aplicación de Windows no se puede empaquetar en Linux. Debe ejecutar PyInstaller en Windows para agrupar una aplicación para Windows)

Página de inicio documentos oficiales

Examples

Instalación y configuración

Pyinstaller es un paquete normal de Python. Se puede instalar utilizando pip:

```
pip install pyinstaller
```

Instalación en Windows
Para Windows, pywin32 o pypiwin32 es un requisito previo. Este último se instala automáticamente cuando se instala pyinstaller usando pip.

Instalación en Mac OS X
PyInstaller funciona con el Python 2.7 predeterminado que se proporciona con el Mac OS X actual. Si se van a usar versiones posteriores de Python o si se deben usar paquetes importantes como PyQT, Numpy, Matplotlib y similares, se recomienda instalarlos usando ya sea MacPorts o Homebrew .

Instalación desde el archivo
Si pip no está disponible, descargue el archivo comprimido desde PyPI .
Para probar la versión de desarrollo, descargue el archivo comprimido desde la rama de

desarrollo de la página de descargas de PyInstaller .

Expanda el archivo y encuentre el script `setup.py` . Ejecute `python setup.py install` con privilegio de administrador para instalar o actualizar PyInstaller.

Verificando la instalación
El comando `pyinstaller` debería existir en la ruta del sistema para todas las plataformas después de una instalación exitosa.
`pyinstaller --version` escribiendo `pyinstaller --version` en la línea de comando. Esto imprimirá la versión actual de pyinstaller.

Usando Pyinstaller

En el caso de uso más simple, simplemente navegue hasta el directorio en el que se encuentra su archivo y escriba:

```
pyinstaller myfile.py
```

Pyinstaller analiza el archivo y crea:

- Un archivo **myfile.spec** en el mismo directorio que `myfile.py`
- Una carpeta de **compilación** en el mismo directorio que `myfile.py`
- Una carpeta **dist** en el mismo directorio que `myfile.py`
- Archivos de registro en la carpeta de **compilación**

La aplicación incluida se puede encontrar en la carpeta **dist**

Opciones
Hay varias opciones que se pueden usar con pyinstaller. Una lista completa de las opciones se puede encontrar aquí .

Una vez que su aplicación se puede empaquetar, abra 'dist \ myfile \ myfile.exe'.

Agrupar en una carpeta

Cuando se usa PyInstaller sin ninguna opción para agrupar `myscript.py` , la salida predeterminada es una única carpeta (denominada `myscript`) que contiene un ejecutable llamado `myscript` (`myscript.exe` en Windows) junto con todas las dependencias necesarias.
La aplicación se puede distribuir comprimiendo la carpeta en un archivo zip.

El modo de una carpeta se puede configurar explícitamente con la opción `-D` o `--onedir`

```
pyinstaller myscript.py -D
```

Ventajas:

Una de las principales ventajas de agrupar en una sola carpeta es que es más fácil depurar problemas. Si alguno de los módulos no se puede importar, puede verificarse inspeccionando la

carpeta.
Otra ventaja se siente durante las actualizaciones. Si hay algunos cambios en el código pero las dependencias utilizadas son *exactamente* las mismas, los distribuidores pueden enviar el archivo ejecutable (que generalmente es más pequeño que la carpeta completa).

Desventajas

La única desventaja de este método es que los usuarios tienen que buscar el ejecutable entre una gran cantidad de archivos.
Además, los usuarios pueden eliminar / modificar otros archivos, lo que puede hacer que la aplicación no funcione correctamente.

Agrupar en un solo archivo

```
pyinstaller myscript.py -F
```

Las opciones para generar un solo archivo son `-F` o `--onefile` . Esto `myscript.exe` el programa en un solo archivo `myscript.exe` .

Un solo archivo ejecutable es más lento que el paquete de una carpeta. También son más difíciles de depurar.

Lea PyInstaller - Distribuir código de Python en línea:
https://riptutorial.com/es/python/topic/2289/pyinstaller---distribuir-codigo-de-python

Capítulo 169: Python Lex-Yacc

Introducción

PLY es una implementación pura de Python de las populares herramientas de construcción de compiladores lex y yacc.

Observaciones

Enlaces adicionales:

1. Documentos oficiales
2. Github

Examples

Empezando con PLY

Para instalar PLY en su máquina para python2 / 3, siga los pasos descritos a continuación:

1. Descarga el código fuente desde aquí .
2. Descomprima el archivo zip descargado.
3. Navegue en la carpeta `ply-3.10` descomprimida
4. Ejecute el siguiente comando en su terminal: `python setup.py install`

Si completó todo lo anterior, ahora debería poder usar el módulo PLY. Puede probarlo abriendo un intérprete de python y escribiendo `import ply.lex` .

Nota: *No* utilice `pip` para instalar PLY, se instalará una distribución rota en su máquina.

El "¡Hola mundo!" de PLY - Una calculadora simple

Demostremos el poder de PLY con un ejemplo simple: este programa tomará una expresión aritmética como una entrada de cadena e intentará resolverla.

Abre tu editor favorito y copia el siguiente código:

```
from ply import lex
import ply.yacc as yacc

tokens = (
    'PLUS',
    'MINUS',
    'TIMES',
    'DIV',
    'LPAREN',
    'RPAREN',
```

```
    'NUMBER',
)

t_ignore = ' \t'

t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIV     = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

def t_NUMBER( t ) :
    r'[0-9]+'
    t.value = int( t.value )
    return t

def t_newline( t ):
  r'\n+'
  t.lexer.lineno += len( t.value )

def t_error( t ):
  print("Invalid Token:",t.value[0])
  t.lexer.skip( 1 )

lexer = lex.lex()

precedence = (
    ( 'left', 'PLUS', 'MINUS' ),
    ( 'left', 'TIMES', 'DIV' ),
    ( 'nonassoc', 'UMINUS' )
)

def p_add( p ) :
    'expr : expr PLUS expr'
    p[0] = p[1] + p[3]

def p_sub( p ) :
    'expr : expr MINUS expr'
    p[0] = p[1] - p[3]

def p_expr2uminus( p ) :
    'expr : MINUS expr %prec UMINUS'
    p[0] = - p[2]

def p_mult_div( p ) :
    '''expr : expr TIMES expr
            | expr DIV expr'''

    if p[2] == '*' :
        p[0] = p[1] * p[3]
    else :
        if p[3] == 0 :
            print("Can't divide by 0")
            raise ZeroDivisionError('integer division by 0')
        p[0] = p[1] / p[3]

def p_expr2NUM( p ) :
    'expr : NUMBER'
    p[0] = p[1]
```

```
def p_parens( p ) :
    'expr : LPAREN expr RPAREN'
    p[0] = p[2]

def p_error( p ):
    print("Syntax error in input!")

parser = yacc.yacc()

res = parser.parse("-4*-(3-5)") # the input
print(res)
```

Guarde este archivo como `calc.py` y ejecútelo.

Salida:

```
-8
```

¿Cuál es la respuesta correcta para `-4 * - (3 - 5)` ?

Parte 1: Tokenizing entrada con Lex

Hay dos pasos que llevó a cabo el código del ejemplo 1: uno fue *tokenizar* la entrada, lo que significa que buscó los símbolos que constituyen la expresión aritmética, y el segundo paso fue *analizar* , lo que implica analizar los tokens extraídos y evaluar el resultado.

Esta sección proporciona un ejemplo simple de cómo *tokenizar* la entrada del usuario, y luego la divide línea por línea.

```
    import ply.lex as lex

    # List of token names. This is always required
    tokens = [
       'NUMBER',
       'PLUS',
       'MINUS',
       'TIMES',
       'DIVIDE',
       'LPAREN',
       'RPAREN',
    ]

    # Regular expression rules for simple tokens
    t_PLUS    = r'\+'
    t_MINUS   = r'-'
    t_TIMES   = r'\*'
    t_DIVIDE  = r'/'
    t_LPAREN  = r'\('
    t_RPAREN  = r'\)'

    # A regular expression rule with some action code
    def t_NUMBER(t):
        r'\d+'
        t.value = int(t.value)
        return t
```

```
    # Define a rule so we can track line numbers
    def t_newline(t):
        r'\n+'
        t.lexer.lineno += len(t.value)

    # A string containing ignored characters (spaces and tabs)
    t_ignore  = ' \t'

    # Error handling rule
    def t_error(t):
        print("Illegal character '%s'" % t.value[0])
        t.lexer.skip(1)

    # Build the lexer
    lexer = lex.lex()

    # Give the lexer some input
    lexer.input(data)

    # Tokenize
    while True:
        tok = lexer.token()
        if not tok:
            break      # No more input
        print(tok)
```

Guarde este archivo como `calclex.py` . Usaremos esto cuando construyamos nuestro analizador de Yacc.

Descompostura

1. Importe el módulo usando `import ply.lex`

2. Todos los lexers deben proporcionar una lista llamada `tokens` que define todos los posibles nombres de token que puede producir el lexer. Esta lista siempre es obligatoria.

```
tokens = [
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
]
```

`tokens` también podrían ser una tupla de cadenas (en lugar de una cadena), donde cada cadena denota un token como antes.

3. La regla de expresiones regulares para cada cadena puede definirse como una cadena o como una función. En cualquier caso, el nombre de la variable debe tener el prefijo t_ para denotar que es una regla para hacer coincidir tokens.

- Para tokens simples, la expresión regular se puede especificar como cadenas: `t_PLUS = r'\+'`

- Si es necesario realizar algún tipo de acción, se puede especificar una regla de token como una función.

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Tenga en cuenta que la regla se especifica como una cadena de documentación dentro de la función. La función acepta un argumento que es una instancia de `LexToken` , realiza alguna acción y luego devuelve el argumento.

Si desea usar una cadena externa como regla de expresión regular para la función en lugar de especificar una cadena de documentación, considere el siguiente ejemplo:

```
@TOKEN(identifier)          # identifier is a string holding the regex
def t_ID(t):
    ...      # actions
```

- Una instancia de objeto `LexToken` (llamemos a este objeto `t`) tiene los siguientes atributos:

    1. `t.type` que es el tipo de token (como una cadena) (por ejemplo: `'NUMBER'` , `'PLUS'` , etc.). Por defecto, `t.type` se establece en el nombre que sigue al prefijo `t_` .
    2. `t.value` que es el lexema (el texto real `t.value`)
    3. `t.lineno` que es el número de línea actual (esto no se actualiza automáticamente, ya que el lexer no sabe nada de los números de línea). Actualiza lineno usando una función llamada `t_newline` .

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

    4. `t.lexpos` que es la posición del token en relación con el comienzo del texto de entrada.

- Si no se devuelve nada de una función de regla de expresiones regulares, el token se descarta. Si desea descartar un token, alternativamente puede agregar el prefijo t_ignore_ a una variable de regla de expresiones regulares en lugar de definir una función para la misma regla.

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # No return value. Token discarded
```

...Es lo mismo que:

```
   t_ignore_COMMENT = r'\#.*'
```

Por supuesto, esto no es válido si está realizando alguna acción cuando ve un comentario. En cuyo caso, use una función para definir la regla de expresiones regulares.

Si no ha definido un token para algunos caracteres pero aún quiere ignorarlo, use `t_ignore = "<characters to ignore>"` (estos prefijos son necesarios):

```
   t_ignore_COMMENT = r'\#.*'
   t_ignore  = ' \t'    # ignores spaces and tabs
```

- Al crear la expresión regular maestra, lex agregará las expresiones regulares especificadas en el archivo de la siguiente manera:

    1. Las fichas definidas por funciones se agregan en el mismo orden en que aparecen en el archivo.
    2. Los tokens definidos por cadenas se agregan en orden decreciente de la longitud de la cadena que define la expresión regular para ese token.

    Si coincide `==` y `=` en el mismo archivo, aproveche estas reglas.

- Los literales son tokens que se devuelven como son. Tanto `t.type` como `t.value` se establecerán en el propio carácter. Defina una lista de literales como tales:

```
literals = [ '+', '-', '*', '/' ]
```

o,

```
literals = "+-*/"
```

Es posible escribir funciones de token que realizan acciones adicionales cuando se combinan literales. Sin embargo, deberás configurar el tipo de token de forma adecuada. Por ejemplo:

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = '{'  # Set token type to the expected literal (ABSOLUTE MUST if this
is a literal)
    return t
```

- Manejar errores con la función t_error.

```
# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1) # skip the illegal token (don't process it)
```

En general, `t.lexer.skip(n)` omite n caracteres en la cadena de entrada.

4. Preparativos finales

   Construya el lexer usando `lexer = lex.lex()` .

   También puede colocar todo dentro de una clase y llamar a la instancia de uso de la clase para definir el lexer. P.ej:

```
 import ply.lex as lex
 class MyLexer(object):
       ...     # everything relating to token rules and error handling comes here as
usual

       # Build the lexer
       def build(self, **kwargs):
           self.lexer = lex.lex(module=self, **kwargs)

       def test(self, data):
           self.lexer.input(data)
           for token in self.lexer.token():
               print(token)

       # Build the lexer and try it out

 m = MyLexer()
 m.build()           # Build the lexer
 m.test("3 + 4")     #
```

   Proporcione entrada utilizando `lexer.input(data)` donde los datos son una cadena

   Para obtener los tokens, use `lexer.token()` que devuelve tokens coincidentes. Puede iterar sobre lexer en un bucle como en:

```
for i in lexer:
    print(i)
```

Parte 2: Análisis de entrada Tokenized con Yacc

Esta sección explica cómo se procesa la entrada tokenizada de la Parte 1: se realiza utilizando las gramáticas libres de contexto (CFG). Se debe especificar la gramática y los tokens se procesan de acuerdo con la gramática. Bajo el capó, el analizador utiliza un analizador LALR.

```
# Yacc example

import ply.yacc as yacc

# Get the token map from the lexer. This is required.
from calclex import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]
```

```
def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

while True:
   try:
       s = raw_input('calc > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
   print(result)
```

Descompostura

- Cada regla gramatical está definida por una función donde la cadena de documentación de esa función contiene la especificación gramatical libre de contexto apropiada. Las declaraciones que conforman el cuerpo de la función implementan las acciones semánticas de la regla. Cada función acepta un solo argumento p que es una secuencia que contiene los valores de cada símbolo gramatical en la regla correspondiente. Los valores de `p[i]` se asignan a símbolos gramaticales como se muestra aquí:

  ```
  def p_expression_plus(p):
      'expression : expression PLUS term'
  ```

```
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

- Para los tokens, el "valor" de la `p[i]` es el mismo que el atributo `p.value` asignado en el módulo lexer. Por lo tanto, `PLUS` tendrá el valor `+` .
- Para no terminales, el valor se determina por lo que se coloque en `p[0]` . Si no se coloca nada, el valor es Ninguno. Además, `p[-1]` no es lo mismo que `p[3]` , ya que `p` no es una lista simple (`p[-1]` puede especificar acciones incrustadas (no se trata aquí)).

Tenga en cuenta que la función puede tener cualquier nombre, siempre que esté precedida por `p_` .

- La `p_error(p)` se define para detectar errores de sintaxis (igual que `yyerror` en yacc / bison).
- Se pueden combinar varias reglas gramaticales en una sola función, lo que es una buena idea si las producciones tienen una estructura similar.

```
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
       term       : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

- Se pueden usar caracteres literales en lugar de fichas.

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
       term       : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

  Por supuesto, los literales deben especificarse en el módulo lexer.

- Las producciones vacías tienen la forma `'''symbol : '''`

- Para establecer explícitamente el símbolo de inicio, use `start = 'foo'` , donde `foo` es algo que no es terminal.
- La configuración de la precedencia y la asociatividad se puede hacer utilizando la variable de precedencia.

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'),  # Nonassociative operators
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),            # Unary minus operator
)
```

  Los tokens se ordenan de menor a mayor precedencia. `nonassoc` significa que esos tokens no se asocian. Esto significa que algo como `a < b < c` es ilegal mientras que `a < b` todavía es legal.
- `parser.out` es un archivo de depuración que se crea cuando se ejecuta el programa yacc por primera vez. Cada vez que se produce un conflicto de desplazamiento / reducción, el analizador siempre cambia.

Lea Python Lex-Yacc en línea: https://riptutorial.com/es/python/topic/10510/python-lex-yacc

Capítulo 170: Python Requests Post

Introducción

Documentación para el módulo de solicitudes de Python en el contexto del método HTTP POST y su función de solicitudes correspondiente

Examples

Post simple

```
from requests import post

foo = post('http://httpbin.org/post', data = {'key':'value'})
```

Realizará una simple operación HTTP POST. Los datos publicados pueden ser en la mayoría de los formatos, sin embargo, los pares de valores clave son los más frecuentes.

Encabezados

Los encabezados pueden ser vistos:

```
print(foo.headers)
```

Un ejemplo de respuesta:

```
{'Content-Length': '439', 'X-Processed-Time': '0.000802993774414', 'X-Powered-By': 'Flask',
'Server': 'meinheld/0.6.1', 'Connection': 'keep-alive', 'Via': '1.1 vegur', 'Access-Control-
Allow-Credentials': 'true', 'Date': 'Sun, 21 May 2017 20:56:05 GMT', 'Access-Control-Allow-
Origin': '*', 'Content-Type': 'application/json'}
```

Los encabezados también se pueden preparar antes de la publicación:

```
headers = {'Cache-Control':'max-age=0',
        'Upgrade-Insecure-Requests':'1',
        'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.99 Safari/537.36',
        'Content-Type':'application/x-www-form-urlencoded',
        'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
        'Referer':'https://www.groupon.com/signup',
        'Accept-Encoding':'gzip, deflate, br',
        'Accept-Language':'es-ES,es;q=0.8'
        }

 foo = post('http://httpbin.org/post', headers=headers, data = {'key':'value'})
```

Codificación

La codificación se puede configurar y ver de la misma manera:

```
 print(foo.encoding)

'utf-8'

foo.encoding = 'ISO-8859-1'
```

Verificación SSL

Las solicitudes por defecto valida los certificados SSL de los dominios. Esto puede ser anulado:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, verify=False)
```

Redireccion

Se seguirá cualquier redirección (por ejemplo, http a https), esto también se puede cambiar:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, allow_redirects=False)
```

Si la operación posterior se ha redirigido, se puede acceder a este valor:

```
print(foo.url)
```

Se puede ver un historial completo de redirecciones:

```
print(foo.history)
```

Formulario de datos codificados

```
from requests import post

payload = {'key1' : 'value1',
           'key2' : 'value2'
           }

foo = post('http://httpbin.org/post', data=payload)
```

Para pasar datos codificados de forma con la operación posterior, los datos deben estructurarse como diccionario y suministrarse como el parámetro de datos.

Si los datos no desean estar codificados en forma, simplemente pase una cadena o un entero al parámetro de datos.

Suministre el diccionario al parámetro json para que Solicitudes formateen los datos automáticamente:

```
from requests import post

payload = {'key1' : 'value1', 'key2' : 'value2'}

foo = post('http://httpbin.org/post', json=payload)
```

Subir archivo

Con el módulo de solicitudes, solo es necesario proporcionar un identificador de archivo en lugar de los contenidos recuperados con `.read()` :

```
from requests import post

files = {'file' : open('data.txt', 'rb')}

foo = post('http://http.org/post', files=files)
```

Nombre de archivo, tipo de contenido y encabezados también se pueden configurar:

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel',
{'Expires': '0'})}

foo = requests.post('http://httpbin.org/post', files=files)
```

Las cadenas también se pueden enviar como un archivo, siempre que se proporcionen como el parámetro de `files` .

Múltiples archivos

Se pueden suministrar varios archivos de la misma manera que un archivo:

```
multiple_files = [
    ('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),
    ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]

foo = post('http://httpbin.org/post', files=multiple_files)
```

Respuestas

Los códigos de respuesta se pueden ver desde una operación posterior:

```
from requests import post

foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.status_code)
```

Datos devueltos

Accediendo a los datos que se devuelven:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.text)
```

Respuestas crudas

En los casos en que necesite acceder al objeto subyacente urllib3 response.HTTPResponse, esto se puede hacer de la siguiente manera:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
res = foo.raw

print(res.read())
```

Autenticación

Autenticación HTTP simple

La autenticación HTTP simple se puede lograr con lo siguiente:

```
from requests import post

foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

Esta es una mano técnicamente corta para lo siguiente:

```
from requests import post
from requests.auth import HTTPBasicAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

Autentificación HTTP Digest

La autenticación HTTP Digest se realiza de una manera muy similar, las solicitudes proporcionan un objeto diferente para esto:

```
from requests import post
from requests.auth import HTTPDigestAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0',
'natas0'))
```

Autenticación personalizada

En algunos casos, los mecanismos de autenticación integrados pueden no ser suficientes, imagine este ejemplo:

Un servidor está configurado para aceptar la autenticación si el remitente tiene la cadena de usuario-agente correcta, un cierto valor de encabezado y proporciona las credenciales correctas a través de la autenticación básica HTTP. Para lograr esto, se debe preparar una clase de autenticación personalizada, subclasificando AuthBase, que es la base para las implementaciones de autenticación de solicitudes:

```
from requests.auth import AuthBase
from requests.auth import _basic_auth_str
from requests._internal_utils import to_native_string

class CustomAuth(AuthBase):

    def __init__(self, secret_header, user_agent , username, password):
        # setup any auth-related data here
```

```
        self.secret_header =  secret_header
        self.user_agent = user_agent
        self.username = username
        self.password = password

    def __call__(self, r):
        # modify and return the request
        r.headers['X-Secret'] = self.secret_header
        r.headers['User-Agent'] = self.user_agent
        r.headers['Authorization'] = _basic_auth_str(self.username, self.password)

        return r
```

Esto puede ser utilizado con el siguiente código:

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

Proxies

Cada operación POST de solicitud puede configurarse para usar proxies de red

Proxies HTTP / S

```
from requests import post

proxies = {
  'http': 'http://192.168.0.128:3128',
  'https': 'http://192.168.0.127:1080',
   }

foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

La autenticación básica HTTP se puede proporcionar de esta manera:

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

SOCKS Proxies

El uso de proxies de calcetines requiere `requests[socks]` dependencias de terceros `requests[socks]` ; una vez instalados, los proxies de calcetines se utilizan de manera muy similar a HTTPBasicAuth:

```
proxies = {
'http': 'socks5://user:pass@host:port',
'https': 'socks5://user:pass@host:port'
}

foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Lea Python Requests Post en línea: https://riptutorial.com/es/python/topic/10021/python-requests-post

Capítulo 171: Python y Excel

Examples

Ponga los datos de la lista en un archivo de Excel.

```
import os, sys
from openpyxl import Workbook
from datetime import datetime

dt = datetime.now()
list_values = [["01/01/2016", "05:00:00", 3], \
               ["01/02/2016", "06:00:00", 4], \
               ["01/03/2016", "07:00:00", 5], \
               ["01/04/2016", "08:00:00", 6], \
               ["01/05/2016", "09:00:00", 7]]

# Create a Workbook on Excel:
wb = Workbook()
sheet = wb.active
sheet.title = 'data'

# Print the titles into Excel Workbook:
row = 1
sheet['A'+str(row)] = 'Date'
sheet['B'+str(row)] = 'Hour'
sheet['C'+str(row)] = 'Value'

# Populate with data
for item in list_values:
    row += 1
    sheet['A'+str(row)] = item[0]
    sheet['B'+str(row)] = item[1]
    sheet['C'+str(row)] = item[2]

# Save a file by date:
filename = 'data_' + dt.strftime("%Y%m%d_%I%M%S") + '.xlsx'
wb.save(filename)

# Open the file for the user:
os.chdir(sys.path[0])
os.system('start excel.exe "%s\\%s"' % (sys.path[0], filename, ))
```

OpenPyXL

OpenPyXL es un módulo para manipular y crear libros de trabajo `xlsx/xlsm/xltx/xltm` en la memoria.

Manipulando y leyendo un libro existente:

```
import openpyxl as opx
#To change an existing wookbook we located it by referencing its path
workbook = opx.load_workbook(workbook_path)
```

`load_workbook()` contiene el parámetro `read_only` , estableciendo esto en `True` cargará el libro como read_only, esto es útil cuando se leen archivos `xlsx` más grandes:

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

Una vez que haya cargado el libro en la memoria, puede acceder a las hojas individuales utilizando `workbook.sheets`

```
first_sheet = workbook.worksheets[0]
```

Si desea especificar el nombre de una hoja disponible, puede usar `workbook.get_sheet_names()` .

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

Finalmente, se puede acceder a las filas de la hoja utilizando `sheet.rows` . Para iterar sobre las filas de una hoja, use:

```
for row in sheet.rows:
    print row[0].value
```

Dado que cada `row` en `rows` es una lista de `Cell` , use `Cell.value` para obtener el contenido de la Celda.

Creando un nuevo libro de ejercicios en la memoria:

```
#Calling the Workbook() function creates a new book in memory
wb = opx.Workbook()

#We can then create a new sheet in the wb
ws = wb.create_sheet('Sheet Name', 0) #0 refers to the index of the sheet order in the wb
```

Se pueden cambiar varias propiedades de la pestaña a través de openpyxl, por ejemplo el `tabColor` :

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

Para guardar nuestro libro creado terminamos con:

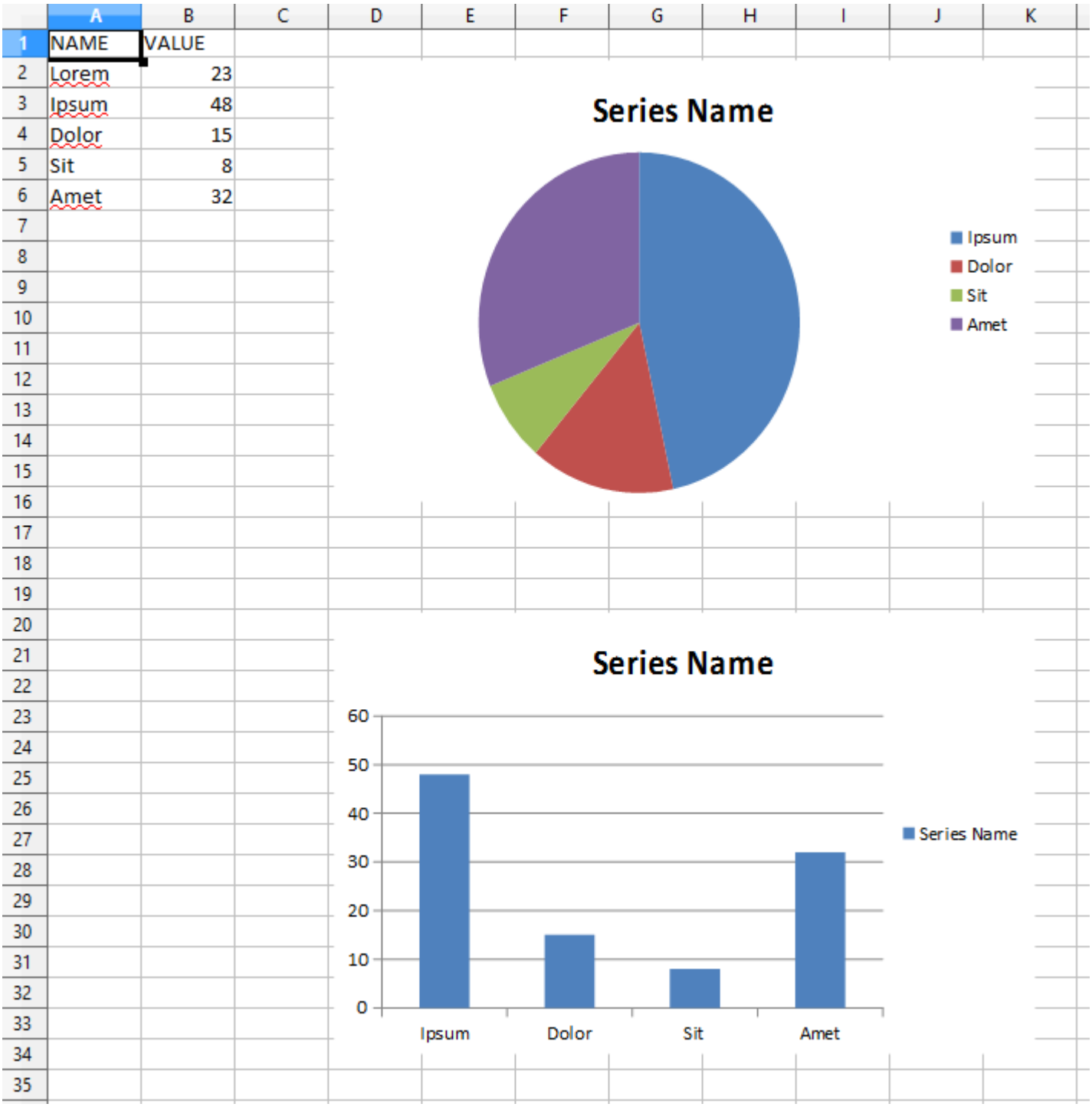
```
wb.save('filename.xlsx')
```

Crear gráficos de Excel con xlsxwriter

```
import xlsxwriter

# sample data
chart_data = [
    {'name': 'Lorem', 'value': 23},
    {'name': 'Ipsum', 'value': 48},
    {'name': 'Dolor', 'value': 15},
```

```
    {'name': 'Sit', 'value': 8},
    {'name': 'Amet', 'value': 32}
]

# excel file path
xls_file = 'chart.xlsx'

# the workbook
workbook = xlsxwriter.Workbook(xls_file)

# add worksheet to workbook
worksheet = workbook.add_worksheet()

row_ = 0
col_ = 0

# write headers
worksheet.write(row_, col_, 'NAME')
col_ += 1
worksheet.write(row_, col_, 'VALUE')
row_ += 1

# write sample data
for item in chart_data:
    col_ = 0
    worksheet.write(row_, col_, item['name'])
    col_ += 1
    worksheet.write(row_, col_, item['value'])
    row_ += 1

# create pie chart
pie_chart = workbook.add_chart({'type': 'pie'})

# add series to pie chart
pie_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})
# insert pie chart
worksheet.insert_chart('D2', pie_chart)

# create column chart
column_chart = workbook.add_chart({'type': 'column'})

# add serie to column chart
column_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})
# insert column chart
worksheet.insert_chart('D20', column_chart)

workbook.close()
```

Resultado:



Lee los datos de excel usando el módulo xlrd

La biblioteca xlrd de Python es para extraer datos de los archivos de hoja de cálculo de Microsoft Excel (tm).

Instalación:-

```
pip install xlrd
```

O puedes usar el archivo setup.py de pypi

https://pypi.python.org/pypi/xlrd

Leyendo una hoja de Excel: - Importe el módulo xlrd y abra el archivo de Excel usando el método open_workbook ().

```
import xlrd
book=xlrd.open_workbook('sample.xlsx')
```

Consultar número de hojas en el excel.

```
print book.nsheets
```

Imprimir los nombres de las hojas

```
print book.sheet_names()
```

Obtener la hoja basada en el índice

```
sheet=book.sheet_by_index(1)
```

Leer el contenido de una celda.

```
cell = sheet.cell(row,col) #where row=row number and col=column number
print cell.value #to print the cell contents
```

Obtenga el número de filas y el número de columnas en una hoja de Excel

```
num_rows=sheet.nrows
num_col=sheet.ncols
```

Obtener hoja de excel por nombre

```
sheets = book.sheet_names()
cur_sheet = book.sheet_by_name(sheets[0])
```

Formato de archivos de Excel con xlsxwriter

```
import xlsxwriter

# create a new file
workbook = xlsxwriter.Workbook('your_file.xlsx')

# add some new formats to be used by the workbook
percent_format = workbook.add_format({'num_format': '0%'})
percent_with_decimal = workbook.add_format({'num_format': '0.0%'})
bold = workbook.add_format({'bold': True})
red_font = workbook.add_format({'font_color': 'red'})
remove_format = workbook.add_format()

# add a new sheet
worksheet = workbook.add_worksheet()
```

```
# set the width of column A
worksheet.set_column('A:A', 30, )

# set column B to 20 and include the percent format we created earlier
worksheet.set_column('B:B', 20, percent_format)

# remove formatting from the first row (change in height=None)
worksheet.set_row('0:0', None, remove_format)

workbook.close()
```

Lea Python y Excel en línea: https://riptutorial.com/es/python/topic/2986/python-y-excel

Capítulo 172: Recolección de basura

Observaciones

En su núcleo, el recolector de basura de Python (a partir de 3.5) es una implementación de conteo de referencia simple. Cada vez que hace una referencia a un objeto (por ejemplo, `a = myobject`) el recuento de referencia en ese objeto (myobject) se incrementa. Cada vez que se elimina una referencia, el recuento de referencias disminuye, y una vez que el recuento de referencias llega a `0` , sabemos que nada tiene una referencia a ese objeto y podemos desasignarlo.

Un malentendido común acerca de cómo funciona la administración de memoria de Python es que la palabra clave `del` delimita la memoria de los objetos. Esto no es verdad. Lo que sucede en realidad es que la palabra clave `del` Delte simplemente reduce los refcount de los objetos, lo que significa que si lo llama suficientes veces para que el refcount llegue a cero, el objeto puede ser recolectado como basura (incluso si en realidad todavía hay referencias al objeto disponible en otra parte de su código).).

Python crea o limpia agresivamente los objetos la primera vez que los necesita. Si realizo la asignación a = object (), la memoria para el objeto se asigna en ese momento (cpython a veces reutilizará ciertos tipos de objetos, por ejemplo, listas bajo el capó, pero, en su mayoría, no mantiene un grupo de objetos libres y realizará la asignación cuando la necesite. De manera similar, tan pronto como el número de ref se reduce a 0, GC lo limpia.

Recolección de basura generacional

En la década de 1960, John McCarthy descubrió una falla fatal en el recuento de basura cuando implementó el algoritmo de recuento de cuentas utilizado por Lisp: ¿Qué sucede si dos objetos se refieren entre sí en una referencia cíclica? ¿Cómo puede alguna vez recolectar esos dos objetos de la basura incluso si no hay referencias externas a ellos si siempre se referirán entre ellos? Este problema también se extiende a cualquier estructura de datos cíclica, como los buffers de un anillo o dos entradas consecutivas en una lista con doble enlace. Python intenta solucionar este problema con un giro ligeramente interesante en otro algoritmo de recolección de basura llamado **Generational Garbage Collection** .

En esencia, cada vez que creas un objeto en Python, lo agrega al final de una lista doblemente enlazada. En ocasiones, Python recorre esta lista, comprueba a qué objetos se refieren los objetos de la lista, y si también están en la lista (veremos por qué podrían no estar en un momento), disminuye aún más sus refcounts. En este punto (en realidad, hay algunas heurísticas que determinan cuándo se mueven las cosas, pero supongamos que después de una sola colección para mantener las cosas simples) cualquier cosa que aún tenga un refcount mayor que 0 se promociona a otra lista vinculada llamada "Generación 1" (esta es la razón por la que no todos los objetos están siempre en la lista de la generación 0) que tiene este bucle aplicado con menos frecuencia. Aquí es donde entra en juego la recolección de basura generacional. Hay 3 generaciones de forma predeterminada en Python (tres listas vinculadas de objetos): La primera

lista (generación 0) contiene todos los objetos nuevos; si ocurre un ciclo GC y los objetos no se recolectan, se mueven a la segunda lista (generación 1), y si ocurre un ciclo GC en la segunda lista y aún no se recolectan, se mueven a la tercera lista (generación 2). La lista de la tercera generación (llamada "generación 2", dado que no tenemos ninguna indexación) es recogida de basura con mucha menos frecuencia que las dos primeras, y la idea es que si su objeto tiene una larga vida útil, no es tan probable que se realice la GCed, y puede que nunca estar en GC durante la vida útil de su aplicación, por lo que no tiene sentido perder el tiempo en cada ejecución del GC. Además, se observa que la mayoría de los objetos se recolectan basura relativamente rápido. De ahora en adelante, llamaremos a estos "buenos objetos" ya que mueren jóvenes. Esto se denomina "hipótesis generacional débil" y también se observó por primera vez en los años 60.

Un lado rápido: a diferencia de las dos primeras generaciones, la lista de tercera generación de larga duración no es recogida de basura en un horario regular. Se verifica cuando la proporción de objetos pendientes de larga duración (aquellos que están en la lista de la tercera generación, pero que aún no han tenido un ciclo GC) con el total de objetos de larga duración en la lista es superior al 25%. Esto se debe a que la tercera lista no tiene límites (las cosas nunca se mueven de ella a otra lista, por lo que solo desaparecen cuando en realidad se recolectan basura), lo que significa que para las aplicaciones en las que está creando muchos objetos de larga duración, los ciclos de GC en la tercera lista puede llegar a ser bastante largo. Al utilizar una relación, logramos "rendimiento lineal amortizado en el número total de objetos"; también conocido, cuanto más larga sea la lista, más tiempo lleva GC, pero con menos frecuencia realizamos GC (aquí está la propuesta original de 2008 para esta heurística de Martin von Löwis para mayor lectura). El acto de realizar una recolección de basura en la tercera generación o lista "madura" se denomina "recolección de basura completa".

Así que la recolección de basura generacional acelera las cosas tremendamente al no requerir que escaneamos objetos que no es probable que necesiten GC todo el tiempo, pero ¿cómo nos ayuda a romper las referencias cíclicas? Probablemente no muy bien, resulta. La función para realmente romper estos ciclos de referencia comienza así :

```
/* Break reference cycles by clearing the containers involved.  This is
 * tricky business as the lists can be changing and we don't know which
 * objects may be freed.  It is possible I screwed something up here.
 */
static void
delete_garbage(PyGC_Head *collectable, PyGC_Head *old)
```

La razón por la que la recolección de basura generacional ayuda con esto es que podemos mantener la longitud de la lista como un conteo separado; cada vez que agregamos un nuevo objeto a la generación incrementamos este conteo, y cada vez que movemos un objeto a otra generación o lo tratamos, decrementamos el conteo. Teóricamente, al final de un ciclo de GC, este recuento (para las primeras dos generaciones de todos modos) siempre debe ser 0. Si no lo es, cualquier cosa que quede en la lista es alguna forma de referencia circular y podemos dejarla. Sin embargo, hay un problema más aquí: ¿Qué pasa si los objetos sobrantes tienen el método mágico de Python `__del__` en ellos? `__del__` se llama cada vez que se destruye un objeto de Python. Sin embargo, si dos objetos en una referencia circular tienen métodos `__del__` , no podemos estar seguros de que destruir uno no romperá el método `__del__` los otros. Para un ejemplo artificial, imagina que escribimos lo siguiente:

```
class A(object):
    def __init__(self, b=None):
        self.b = b

    def __del__(self):
        print("We're deleting an instance of A containing:", self.b)

class B(object):
    def __init__(self, a=None):
        self.a = a

    def __del__(self):
        print("We're deleting an instance of B containing:", self.a)
```

y establecemos una instancia de A y una instancia de B para que apunten entre sí y luego terminan en el mismo ciclo de recolección de basura? Digamos que elegimos uno al azar y desechamos nuestra instancia de A primero; Se `__del__` método `__del__` de A, se imprimirá y luego A se liberará. Luego llegamos a B, llamamos a su método `__del__` , y ¡ `__del__` ! Segfault! A ya no existe. Podríamos arreglar esto llamando primero a `__del__` métodos `__del__` que `__del__` , luego haciendo otra pasada para repartir todo, sin embargo, esto introduce otro problema: ¿Qué pasa si uno de los `__del__` método `__del__` guarda una referencia del otro objeto que está a punto de ser GCed y ¿Tiene una referencia a nosotros en otro lugar? Todavía tenemos un ciclo de referencia, pero ahora no es posible hacer GC en ninguno de los dos objetos, incluso si ya no están en uso. Tenga en cuenta que incluso si un objeto no es parte de una estructura de datos circular, podría revivir en su propio método `__del__` ; Python tiene una verificación de esto y detendrá la GCing si un refcount de objetos ha aumentado después de que se haya llamado a su método `__del__` .

CPython se ocupa de esto al pegar esos objetos `__del__` de GC (cualquier cosa con algún tipo de referencia circular y un método `__del__`) en una lista global de basura no recolectable y luego dejarla ahí por toda la eternidad:

```
/* list of uncollectable objects */
static PyObject *garbage = NULL;
```

Examples

Recuento de referencias

La gran mayoría de la administración de memoria de Python se maneja con conteo de referencias.

Cada vez que se hace referencia a un objeto (por ejemplo, asignado a una variable), su recuento de referencia aumenta automáticamente. Cuando se elimina la referencia (por ejemplo, la variable queda fuera del alcance), su recuento de referencia se reduce automáticamente.

Cuando el recuento de referencia llega a cero, el objeto se **destruye inmediatamente** y la memoria se libera inmediatamente. Por lo tanto, para la mayoría de los casos, el recolector de basura ni siquiera es necesario.

```
>>> import gc; gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def foo():
        Track()
        # destructed immediately since no longer has any references
        print("---")
        t = Track()
        # variable is referenced, so it's not destructed yet
        print("---")
        # variable is destructed when function exits
>>> foo()
Initialized
Destructed
---
Initialized
---
Destructed
```

Para demostrar aún más el concepto de referencias:

```
>>> def bar():
        return Track()
>>> t = bar()
Initialized
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> t = None          # not destructed yet - another_t still refers to it
>>> another_t = None  # final reference gone, object is destructed
Destructed
```

Recolector de basura para ciclos de referencia

La única vez que se necesita el recolector de basura es si tiene un *ciclo de referencia* . El ejemplo simple de un ciclo de referencia es uno en el que A se refiere a B y B se refiere a A, mientras que nada más se refiere a A o B. No se puede acceder a A ni a B desde cualquier lugar del programa, por lo que se pueden destruir de forma segura. sin embargo, sus recuentos de referencia son 1 y, por lo tanto, no se pueden liberar únicamente con el algoritmo de recuento de referencia.

```
>>> import gc; gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> A = Track()
Initialized
>>> B = Track()
Initialized
>>> A.other = B
>>> B.other = A
>>> del A; del B  # objects are not destructed due to reference cycle
```

```
>>> gc.collect()  # trigger collection
Destructed
Destructed
4
```

Un ciclo de referencia puede ser arbitrario largo. Si A apunta a B apunta a C apunta a ... apunta a Z que apunta a A, entonces no se recolectarán A a Z, hasta la fase de recolección de basura:

```
>>> objs = [Track() for _ in range(10)]
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
>>> for i in range(len(objs)-1):
...     objs[i].other = objs[i + 1]
...
>>> objs[-1].other = objs[0]  # complete the cycle
>>> del objs                  # no one can refer to objs now - still not destructed
>>> gc.collect()
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
20
```

Efectos del comando del

Eliminar un nombre de variable del ámbito usando `del v` , o eliminar un objeto de una colección usando `del v[item]` o `del[i:j]` , o eliminar un atributo usando `del v.name` , o cualquier otra forma de eliminar referencias a un objeto, *no* desencadena ninguna llamada de destructor ni ninguna memoria liberada en sí misma. Los objetos solo se destruyen cuando su cuenta de referencia llega a cero.

```
>>> import gc
>>> gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def bar():
    return Track()
>>> t = bar()
Initialized
```

```
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> del t          # not destructed yet - another_t still refers to it
>>> del another_t  # final reference gone, object is destructed
Destructed
```

Reutilización de objetos primitivos.

Una cosa interesante a tener en cuenta que puede ayudar a optimizar sus aplicaciones es que las primitivas también se vuelven a contar bajo el capó. Echemos un vistazo a los números; para todos los enteros entre -5 y 256, Python siempre reutiliza el mismo objeto:

```
>>> import sys
>>> sys.getrefcount(1)
797
>>> a = 1
>>> b = 1
>>> sys.getrefcount(1)
799
```

Tenga en cuenta que refcount aumenta, lo que significa que `a` y `b` referencia al mismo objeto subyacente cuando se refieren a la primitiva `1` . Sin embargo, para números más grandes, Python en realidad no reutiliza el objeto subyacente:

```
>>> a = 999999999
>>> sys.getrefcount(999999999)
3
>>> b = 999999999
>>> sys.getrefcount(999999999)
3
```

Debido a que el refcount de `999999999` no cambia cuando se asigna a `a` y `b` se puede inferir que se refieren a dos objetos subyacentes diferentes, aunque ambos se les asigna el mismo primitivo.

Viendo el refcount de un objeto

```
>>> import sys
>>> a = object()
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

Forzar la desasignación de objetos.

Puede forzar la desasignación de objetos incluso si su refcount no es 0 en Python 2 y 3.

Ambas versiones utilizan el módulo `ctypes` para hacerlo.

ADVERTENCIA: haciendo esto *dejará* su entorno Python inestable y propenso a estrellarse sin un rastreo! El uso de este método también podría introducir problemas de seguridad (bastante improbable). Desasigne solo los objetos que está seguro de que nunca volverá a hacer referencia. Siempre.

Python 3.x 3.0

```
import ctypes
deallocated = 12345
ctypes.pythonapi._Py_Dealloc(ctypes.py_object(deallocated))
```

Python 2.x 2.3

```
import ctypes, sys
deallocated = 12345
(ctypes.c_char * sys.getsizeof(deallocated)).from_address(id(deallocated))[:4] = '\x00' * 4
```

Después de ejecutar, cualquier referencia al objeto ahora desasignado hará que Python produzca un comportamiento indefinido o se bloquee, sin un rastreo. Probablemente hubo una razón por la que el recolector de basura no eliminó ese objeto ...

Si desasigna `None` , aparece un mensaje especial: `Fatal Python error: deallocating None` antes de que se bloquee.

Gestionando la recogida de basura.

Hay dos enfoques para influir cuando se realiza una limpieza de memoria. Influyen en la frecuencia con la que se realiza el proceso automático y el otro está activando manualmente una limpieza.

El recolector de basura se puede manipular ajustando los umbrales de recolección que afectan la frecuencia a la que se ejecuta el recolector. Python utiliza un sistema de gestión de memoria basado en la generación. Los nuevos objetos se guardan en la generación más nueva - **generación0** y con cada colección sobrevivida, los objetos se promueven a las generaciones anteriores. Después de llegar a la última generación - **generación2** , ya no se promocionan.

Los umbrales se pueden cambiar usando el siguiente fragmento de código:

```
import gc
gc.set_threshold(1000, 100, 10) # Values are just for demonstration purpose
```

El primer argumento representa el umbral para recolectar **generation0** . Cada vez que el número de **asignaciones** supera el número de **desasignaciones** por 1000, se llamará al recolector de basura.

Las generaciones anteriores no se limpian en cada ejecución para optimizar el proceso. El segundo y tercer argumento son **opcionales** y controlan la frecuencia con la que se limpian las generaciones anteriores. Si la **generación0** se procesó 100 veces sin limpiar la **generación1** ,

entonces se procesará la **generación1** . De manera similar, los objetos en la **generación 2** se procesarán solo cuando los de la **generación 1 se** hayan limpiado 10 veces sin tocar la **generación 2** .

Una instancia en la que es beneficioso establecer manualmente los umbrales es cuando el programa asigna una gran cantidad de objetos pequeños sin desasignarlos, lo que hace que el recolector de basura se ejecute con demasiada frecuencia (cada una de **las** asignaciones de objetos de umbral de **generación**). A pesar de que el colector es bastante rápido, cuando se ejecuta en una gran cantidad de objetos plantea un problema de rendimiento. De todos modos, no hay una estrategia única para todos los límites para elegir los umbrales y es confiable en cada caso de uso.

La activación manual de una colección se puede hacer como en el siguiente fragmento de código:

```
import gc
gc.collect()
```

La recolección de basura se activa automáticamente en función del número de asignaciones y desasignaciones, no en la memoria consumida o disponible. En consecuencia, cuando se trabaja con objetos grandes, la memoria puede agotarse antes de que se active la limpieza automática. Esto es un buen caso de uso para llamar manualmente al recolector de basura.

Aunque es posible, no es una práctica recomendada. Evitar las pérdidas de memoria es la mejor opción. De todos modos, en grandes proyectos, detectar la pérdida de memoria puede ser una tarea fácil y activar manualmente una recolección de basura se puede usar como una solución rápida hasta una depuración adicional.

Para los programas de larga duración, la recolección de basura puede activarse en una base de tiempo o evento. Un ejemplo para el primero es un servidor web que activa una colección después de un número fijo de solicitudes. Para más adelante, un servidor web que activa una recolección de basura cuando se recibe un cierto tipo de solicitud.

No espere a que la recolección de basura se limpie

El hecho de que la recolección de basura se limpie no significa que deba esperar a que se limpie el ciclo de recolección de basura.

En particular, no debe esperar a que la recolección de basura cierre los manejadores de archivos, las conexiones de base de datos y las conexiones de red abiertas.

por ejemplo:

En el siguiente código, asume que el archivo se cerrará en el siguiente ciclo de recolección de basura, si f fue la última referencia al archivo.

```
>>> f = open("test.txt")
>>> del f
```

Una forma más explícita de limpiar es llamar a `f.close()` . Puede hacerlo aún más elegante, es

decir, utilizando la instrucción `with` , también conocida como administrador de contexto :

```
>>> with open("test.txt") as f:
...     pass
...     # do something with f
>>> #now the f object still exists, but it is closed
```

La instrucción `with` permite sangrar su código debajo del archivo abierto. Esto hace que sea explícito y más fácil ver cuánto tiempo se mantiene abierto un archivo. También siempre cierra un archivo, incluso si se produce una excepción en el bloque `while` .

Lea Recolección de basura en línea: https://riptutorial.com/es/python/topic/2532/recoleccion-de-basura

Capítulo 173: Reconocimiento óptico de caracteres

Introducción

El reconocimiento óptico de caracteres es convertir imágenes de texto en texto real. En estos ejemplos, encuentre formas de usar OCR en python.

Examples

PyTesseract

PyTesseract es un paquete de Python en desarrollo para OCR.

Usar PyTesseract es bastante fácil:

```
try:
    import Image
except ImportError:
    from PIL import Image

import pytesseract

    #Basic OCR
    print(pytesseract.image_to_string(Image.open('test.png')))

    #In French
    print(pytesseract.image_to_string(Image.open('test-european.jpg'), lang='fra'))
```

PyTesseract es de código abierto y se puede encontrar aquí .

PyOCR

Otro módulo de algún uso es `PyOCR` , cuyo código fuente está aquí .

También es fácil de usar y tiene más funciones que `PyTesseract` .

Para inicializar:

```
from PIL import Image
import sys

import pyocr
import pyocr.builders

tools = pyocr.get_available_tools()
# The tools are returned in the recommended order of usage
tool = tools[0]
```

```
langs = tool.get_available_languages()
lang = langs[0]
# Note that languages are NOT sorted in any way. Please refer
# to the system locale settings for the default language
# to use.
```

Y algunos ejemplos de uso:

```
txt = tool.image_to_string(
    Image.open('test.png'),
    lang=lang,
    builder=pyocr.builders.TextBuilder()
)
# txt is a Python string

word_boxes = tool.image_to_string(
    Image.open('test.png'),
    lang="eng",
    builder=pyocr.builders.WordBoxBuilder()
)
# list of box objects. For each box object:
#   box.content is the word in the box
#   box.position is its position on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

line_and_word_boxes = tool.image_to_string(
    Image.open('test.png'), lang="fra",
    builder=pyocr.builders.LineBoxBuilder()
)
# list of line objects. For each line object:
#   line.word_boxes is a list of word boxes (the individual words in the line)
#   line.content is the whole text of the line
#   line.position is the position of the whole line on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

# Digits - Only Tesseract (not 'libtesseract' yet !)
digits = tool.image_to_string(
    Image.open('test-digits.png'),
    lang=lang,
    builder=pyocr.tesseract.DigitBuilder()
)
# digits is a python string
```

Lea Reconocimiento óptico de caracteres en línea:
https://riptutorial.com/es/python/topic/9302/reconocimiento-optico-de-caracteres

Capítulo 174: Recursion

Observaciones

La recursión necesita una condición de parada `stopCondition` para salir de la recursión.

La variable original se debe pasar a la función recursiva para que se almacene.

Examples

Suma de números del 1 al n

Si quisiera averiguar la suma de los números del `1` al `n` donde `n` es un número natural, puedo hacer `1 + 2 + 3 + 4 + ... + (several hours later) + n` . Alternativamente, podría escribir un bucle `for` :

```
n = 0
for i in range (1, n+1):
    n += i
```

O podría usar una técnica conocida como recursión:

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

La recursión tiene ventajas sobre los dos métodos anteriores. La recursión toma menos tiempo que escribir `1 + 2 + 3` para una suma de 1 a 3. Para la `recursion(4)` , la recursión se puede usar para trabajar hacia atrás:

Llamadas a funciones: (4 -> 4 + 3 -> 4 + 3 + 2 -> 4 + 3 + 2 + 1 -> 10)

Mientras que el bucle `for` está trabajando estrictamente hacia adelante: (1 -> 1 + 2 -> 1 + 2 + 3 -> 1 + 2 + 3 + 4 -> 10). A veces, la solución recursiva es más simple que la solución iterativa. Esto es evidente cuando se implementa una reversión de una lista vinculada.

El qué, cómo y cuándo de la recursión

La recursión se produce cuando una llamada de función hace que esa misma función se vuelva a llamar antes de que finalice la llamada de función original. Por ejemplo, considere la expresión matemática conocida `x!` (Es decir, la operación factorial). La operación factorial se define para todos los enteros no negativos de la siguiente manera:

- Si el número es 0, entonces la respuesta es 1.
- De lo contrario, la respuesta es que el número multiplicado por el factorial es uno menos

que ese número.

En Python, una implementación ingenua de la operación factorial se puede definir como una función de la siguiente manera:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Las funciones de recursión pueden ser difíciles de comprender a veces, así que veamos esto paso a paso. Considera la expresión `factorial(3)` . Esta y *todas* las llamadas a funciones crean un nuevo **entorno** . Un entorno es básicamente una tabla que asigna identificadores (por ejemplo, `n` , `factorial` , `print` , etc.) a sus valores correspondientes. En cualquier momento, puede acceder al entorno actual utilizando `locals()` . En la primera llamada a la función, la única variable local que se define es `n = 3` . Por lo tanto, la impresión de `locals()` mostraría `{'n': 3}` . Como `n == 3` , el valor de retorno se convierte en `n * factorial(n - 1)` .

En este siguiente paso es donde las cosas pueden ser un poco confusas. Mirando nuestra nueva expresión, ya sabemos qué es `n` . Sin embargo, todavía no sabemos qué `factorial(n - 1)` es. Primero, `n - 1` evalúa a `2` . Entonces, `2` se pasa a `factorial` como el valor para `n` . Dado que se trata de una nueva llamada de función, se crea un segundo entorno para almacenar esta nueva `n` . Sea *A* el primer entorno y *B* el segundo entorno. *A* todavía existe y es igual a `{'n': 3}` , sin embargo, *B* (que es igual a `{'n': 2}`) es el entorno actual. Al observar el cuerpo de la función, el valor de retorno es, nuevamente, `n * factorial(n - 1)` . Sin evaluar esta expresión, vamos a sustituirla en la expresión de retorno original. Al hacer esto, estamos descartando mentalmente *B* , así que recuerde sustituir `n` consecuencia (es decir, las referencias a *B* 's `n` se reemplazan con `n - 1` que usa *A* ' s `n`). Ahora, la expresión de retorno original se convierte en `n * ((n - 1) * factorial((n - 1) - 1))` . Tome un segundo para asegurarse de que entiende por qué esto es así.

Ahora, evaluemos la porción `factorial((n - 1) - 1))` de eso. Como *A* 's `n == 3` , estamos pasando `1` a `factorial` . Por lo tanto, estamos creando un nuevo entorno *C* que es igual a `{'n': 1}` . Una vez más, el valor de retorno es `n * factorial(n - 1)` . Entonces, reemplacemos `factorial((n - 1) - 1))` de la expresión de retorno "original" de manera similar a como ajustamos la expresión de retorno original anteriormente. La expresión "original" ahora es `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))` .

Casi termino. Ahora, necesitamos evaluar `factorial((n - 2) - 1)` . Esta vez, estamos pasando en `0` . Por lo tanto, esto se evalúa a `1` . Ahora, vamos a realizar nuestra última sustitución. La expresión de retorno "original" ahora es `n * ((n - 1) * ((n - 2) * 1))` . Recordando que la expresión de retorno original se evalúa en *A* , la expresión se convierte en `3 * ((3 - 1) * ((3 - 2) * 1))` . Esto, por supuesto, se evalúa a 6. Para confirmar que esta es la respuesta correcta, ¡recuerde que `3! == 3 * 2 * 1 == 6` . Antes de seguir leyendo, asegúrese de comprender completamente el concepto de entornos y cómo se aplican a la recursión.

La declaración `if n == 0: return 1` se llama un caso base. Esto se debe a que no exhibe recursión. Un caso base es absolutamente necesario. Sin uno, te encontrarás con una recursión

infinita. Dicho esto, siempre que tenga al menos un caso base, puede tener tantos casos como desee. Por ejemplo, podríamos haber escrito un `factorial` equivalente de la siguiente manera:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

También puede tener varios casos de recursión, pero no vamos a entrar en eso, ya que es relativamente poco frecuente y, a menudo, es difícil de procesar mentalmente.

También puede tener llamadas de función recursiva "paralelas". Por ejemplo, considere la secuencia de Fibonacci que se define de la siguiente manera:

- Si el número es 0, entonces la respuesta es 0.
- Si el número es 1, entonces la respuesta es 1.
- De lo contrario, la respuesta es la suma de los dos números anteriores de Fibonacci.

Podemos definir esto como sigue:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

No recorreré esta función tan a fondo como lo hice con `factorial(3)` , pero el valor de retorno final de `fib(5)` es equivalente a la siguiente expresión (*sintácticamente* no válida):

```
(
  fib((n - 2) - 2)
  +
  (
    fib(((n - 2) - 1) - 2)
    +
    fib(((n - 2) - 1) - 1)
  )
)
+
(
  (
    fib(((n - 1) - 2) - 2)
    +
    fib(((n - 1) - 2) - 1)
  )
  +
  (
    fib(((n - 1) - 1) - 2)
    +
    (
      fib((((n - 1) - 1) - 1) - 2)
      +
```

```
      fib((((n - 1) - 1) - 1) - 1)
    )
  )
)
```

Esto se convierte en `(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))` que, por supuesto, se evalúa en `5` .

Ahora, vamos a cubrir algunos términos más de vocabulario:

- Una **llamada de cola** es simplemente una llamada de función recursiva que es la última operación que se realiza antes de devolver un valor. Para ser claros, `return foo(n - 1)` es una llamada de cola, pero `return foo(n - 1) + 1` no (ya que la adición es la última operación).
- **La optimización de llamadas de cola** (TCO) es una forma de reducir automáticamente la recursión en funciones recursivas.
- **La eliminación de la llamada de cola** (TCE) es la reducción de una llamada de la cola a una expresión que se puede evaluar sin recursión. TCE es un tipo de TCO.

La optimización de llamadas de cola es útil por varias razones:

- El intérprete puede minimizar la cantidad de memoria ocupada por los entornos. Como ninguna computadora tiene memoria ilimitada, las llamadas excesivas a funciones recursivas conducirían a un **desbordamiento de pila** .
- El intérprete puede reducir el número de interruptores de **cuadros** de **pila** .

Python no tiene una forma de TCO implementada por varias razones . Por lo tanto, se requieren otras técnicas para evitar esta limitación. El método de elección depende del caso de uso. Con algo de intuición, las definiciones de `factorial` y `fib` se pueden convertir de manera relativamente fácil a código iterativo de la siguiente manera:

```
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product

def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

Esta suele ser la forma más eficiente de eliminar manualmente la recursión, pero puede ser bastante difícil para funciones más complejas.

Otra herramienta útil es el decorador lru_cache de Python, que se puede usar para reducir el número de cálculos redundantes.

Ahora tiene una idea de cómo evitar la recursión en Python, pero ¿cuándo *debería* usar la recursión? La respuesta es "no a menudo". Todas las funciones recursivas pueden ser

implementadas iterativamente. Es simplemente una cuestión de averiguar cómo hacerlo. Sin embargo, hay casos raros en los que la recursión está bien. La recursión es común en Python cuando las entradas esperadas no causan un número significativo de llamadas a una función recursiva.

Si la recursión es un tema que le interesa, le imploro que estudie lenguajes funcionales como Scheme o Haskell. En tales idiomas, la recursión es mucho más útil.

Tenga en cuenta que el ejemplo anterior para la secuencia de Fibonacci, aunque es bueno para mostrar cómo aplicar la definición en python y el uso posterior de la caché lru, tiene un tiempo de ejecución ineficiente ya que realiza 2 llamadas recursivas para cada caso no básico. El número de llamadas a la función crece exponencialmente a `n` .
Más bien, de forma no intuitiva, una implementación más eficiente usaría la recursión lineal:

```
def fib(n):
    if n <= 1:
        return (n,0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

Pero ese tiene el problema de devolver un *par* de números. Esto enfatiza que algunas funciones realmente no ganan mucho con la recursión.

Exploración de árboles con recursión

Digamos que tenemos el siguiente árbol:

```
root
- A
  - AA
  - AB
- B
  - BA
  - BB
    - BBA
```

Ahora, si deseamos enumerar todos los nombres de los elementos, podríamos hacer esto con un simple bucle for. Suponemos que hay una función `get_name()` para devolver una cadena del nombre de un nodo, una función `get_children()` para devolver una lista de todos los `get_children()` de un nodo dado en el árbol, y una función `get_root()` para obtener el nodo raíz

```
root = get_root(tree)
for node in get_children(root):
    print(get_name(node))
    for child in get_children(node):
        print(get_name(child))
        for grand_child in get_children(child):
            print(get_name(grand_child))
# prints: A, AA, AB, B, BA, BB, BBA
```

Esto funciona bien y rápido, pero ¿qué pasa si los subnodos tienen subnodos propios? Y esos

subnodos podrían tener más subnodos ... ¿Qué pasa si no sabe de antemano cuántos habrá? Un método para resolver esto es el uso de la recursión.

```
def list_tree_names(node):
    for child in get_children(node):
        print(get_name(child))
        list_tree_names(node=child)

list_tree_names(node=get_root(tree))
# prints: A, AA, AB, B, BA, BB, BBA
```

Quizás desee no imprimir, pero devuelva una lista plana de todos los nombres de nodo. Esto se puede hacer pasando una lista móvil como parámetro.

```
def list_tree_names(node, lst=[]):
    for child in get_children(node):
        lst.append(get_name(child))
        list_tree_names(node=child, lst=lst)
    return lst

list_tree_names(node=get_root(tree))
# returns ['A', 'AA', 'AB', 'B', 'BA', 'BB', 'BBA']
```

Incrementando la profundidad máxima de recursión

Hay un límite a la profundidad de la posible recursión, que depende de la implementación de Python. Cuando se alcanza el límite, se genera una excepción RuntimeError:

```
RuntimeError: Maximum Recursion Depth Exceeded
```

Aquí hay una muestra de un programa que causaría este error:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))
cursing(0)
# Out: I recursed 1083 times!
```

Es posible cambiar el límite de profundidad de recursión usando

```
sys.setrecursionlimit(limit)
```

Puede verificar cuáles son los parámetros actuales del límite ejecutando:

```
sys.getrecursionlimit()
```

Ejecutando el mismo método anterior con nuestro nuevo límite obtenemos

```
sys.setrecursionlimit(2000)
```

```
cursing(0)
# Out: I recursed 1997 times!
```

Desde Python 3.5, la excepción es un RecursionError, que se deriva de RuntimeError.

Recursión de cola - Mala práctica

Cuando lo único que se devuelve de una función es una llamada recursiva, se la denomina recursión de cola.

Aquí hay un ejemplo de cuenta regresiva escrita usando la recursión de la cola:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Cualquier cálculo que se pueda hacer usando la iteración también se puede hacer usando la recursión. Aquí hay una versión de find_max escrita usando la recursión de la cola:

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

La recursión de cola se considera una mala práctica en Python, ya que el compilador de Python no maneja la optimización para las llamadas recursivas de cola. La solución recursiva en casos como este utiliza más recursos del sistema que la solución iterativa equivalente.

Optimización de la recursión de cola a través de la introspección de la pila

Por defecto, la pila de recursión de Python no puede exceder los 1000 cuadros. Esto se puede cambiar configurando `sys.setrecursionlimit(15000)` que es más rápido, sin embargo, este método consume más memoria. En su lugar, también podemos resolver el problema de recursión de cola utilizando la introspección de pila.

```
#!/usr/bin/env python2.4
# This program shows off a python decorator which implements tail call optimization. It
# does this by throwing an exception if it is it's own grandparent, and catching such
# exceptions to recall the stack.

import sys

class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs
```

```
def tail_call_optimized(g):
"""
This function decorates a function with tail call
optimization. It does this by throwing an exception
if it is it's own grandparent, and catching such
exceptions to fake the tail call optimization.

This function fails if the decorated
function recurses in a non-tail context.
"""

    def func(*args, **kwargs):
        f = sys._getframe()
        if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
            raise TailRecurseException(args, kwargs)
        else:
            while 1:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException, e:
                    args = e.args
                    kwargs = e.kwargs
    func.__doc__ = g.__doc__
    return func
```

Para optimizar las funciones recursivas, podemos usar el decorador `@tail_call_optimized` para llamar a nuestra función. Aquí hay algunos de los ejemplos comunes de recursión que utilizan el decorador descrito anteriormente:

Ejemplo factorial:

```
@tail_call_optimized
def factorial(n, acc=1):
  "calculate a factorial"
  if n == 0:
    return acc
  return factorial(n-1, n*acc)

print factorial(10000)
# prints a big, big number,
# but doesn't hit the recursion limit.
```

Ejemplo de Fibonacci:

```
@tail_call_optimized
def fib(i, current = 0, next = 1):
  if i == 0:
    return current
  else:
    return fib(i - 1, next, current + next)

print fib(10000)
# also prints a big number,
# but doesn't hit the recursion limit.
```

Lea Recursion en línea: https://riptutorial.com/es/python/topic/1716/recursion

Capítulo 175: Redes Python

Observaciones

Ejemplo de socket de cliente Python (muy) básico

Examples

El ejemplo más simple de cliente / servidor de socket de Python.

Lado del servidor:

```
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8089))
serversocket.listen(5) # become a server socket, maximum 5 connections

while True:
    connection, address = serversocket.accept()
    buf = connection.recv(64)
    if len(buf) > 0:
        print(buf)
    break
```

Lado del cliente:

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')
```

Primero ejecute el SocketServer.py, y asegúrese de que el servidor esté listo para escuchar / recibir algo. Luego, el cliente envía la información al servidor; Después de que el servidor recibió algo, termina

Creando un servidor HTTP simple

Para compartir archivos o alojar sitios web simples (http y javascript) en su red local, puede usar el módulo SimpleHTTPServer integrado de Python. Python debería estar en tu variable Path. Vaya a la carpeta donde están sus archivos y escriba:

Para `python 2` :

```
$ python -m SimpleHTTPServer <portnumber>
```

Para `python 3` :

```
$ python3 -m http.server <portnumber>
```

Si no se da el número de puerto, `8000` es el puerto predeterminado. Así que la salida será:

> Sirviendo HTTP en el puerto 0.0.0.0 8000 ...

Puede acceder a sus archivos a través de cualquier dispositivo conectado a la red local escribiendo `http://hostipaddress:8000/` .

`hostipaddress` es su dirección IP local que probablemente comienza con `192.168.xx`

Para finalizar el módulo simplemente presione `ctrl+c`.

Creando un servidor TCP

Puede crear un servidor TCP utilizando la biblioteca `socketserver` . Aquí hay un servidor de eco simple.

Lado del servidor

```
from sockerserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('connection from:', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    server = TCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Lado del cliente

```
from socket import socket, AF_INET, SOCK_STREAM
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 5000))
sock.send(b'Monty Python')
sock.recv(8192)  # returns b'Monty Python'
```

`socketserver` hace que sea relativamente fácil crear servidores TCP simples. Sin embargo, debe tener en cuenta que, de forma predeterminada, los servidores son de un solo hilo y solo pueden atender a un cliente a la vez. Si desea manejar múltiples clientes, cree una instancia de `ThreadingTCPServer` lugar.

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Creando un Servidor UDP

Un servidor UDP se crea fácilmente utilizando la biblioteca de `socketserver` .

un servidor de tiempo simple:

```
import time
from socketserver import BaseRequestHandler, UDPServer

class CtimeHandler(BaseRequestHandler):
    def handle(self):
    print('connection from: ', self.client_address)
    # Get message and client socket
    msg, sock = self.request
    resp = time.ctime()
    sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    server = UDPServer(('', 5000), CtimeHandler)
    server.serve_forever()
```

Pruebas:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> sock = socket(AF_INET, SOCK_DGRAM)
>>> sick.sendto(b'', ('localhost', 5000))
0
>>> sock.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 5000))
```

Inicie Simple HttpServer en un hilo y abra el navegador

Útil si tu programa está generando páginas web a lo largo del camino.

```
from http.server import HTTPServer, CGIHTTPRequestHandler
import webbrowser
import threading

def start_server(path, port=8000):
    '''Start a simple webserver serving path on port'''
    os.chdir(path)
    httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
    httpd.serve_forever()

# Start the server in a new thread
port = 8000
daemon = threading.Thread(name='daemon_server',
                          target=start_server,
                          args=('.', port)
daemon.setDaemon(True) # Set as a daemon so it will be killed once the main thread is dead.
daemon.start()

# Open the web browser
webbrowser.open('http://localhost:{}'.format(port))
```

Lea Redes Python en línea: https://riptutorial.com/es/python/topic/1309/redes-python

Capítulo 176: Reducir

Sintaxis

- reducir (función, iterable [, inicializador])

Parámetros

Parámetro	Detalles
función	Función que se utiliza para reducir lo iterable (debe tomar dos argumentos). (*solo posicional*)
iterable	iterable que va a ser reducido. (*solo posicional*)
inicializador	Valor de inicio de la reducción. (*opcional* , *solo posicional*)

Observaciones

`reduce` podría no ser siempre la función más eficiente. Para algunos tipos hay funciones o métodos equivalentes:

- `sum()` para la suma de una secuencia que contiene elementos *sumables* (no cadenas):

```
sum([1,2,3])                                    # = 6
```

- `str.join` para la concatenación de cuerdas:

```
''.join(['Hello', ',', ' World'])               # = 'Hello, World'
```

- `next` junto con un generador podría ser una variante de cortocircuito en comparación con `reduce` :

```
# First falsy item:
next((i for i in [100, [], 20, 0] if not i)) # = []
```

Examples

Visión general

```
# No import needed


# No import required...
```

```
from functools import reduce # ... but it can be loaded from the functools module


from functools import reduce # mandatory
```

`reduce` reduce un iterable aplicando una función repetidamente en el siguiente elemento de un resultado `iterable` y acumulativo hasta el momento.

```
def add(s1, s2):
    return s1 + s2

asequence = [1, 2, 3]

reduce(add, asequence)  # equivalent to: add(add(1,2),3)
# Out: 6
```

En este ejemplo, definimos nuestra propia función de `add` . Sin embargo, Python viene con una función equivalente estándar en el módulo del `operator` :

```
import operator
reduce(operator.add, asequence)
# Out: 6
```

`reduce` también se puede pasar un valor inicial:

```
reduce(add, asequence, 10)
# Out: 16
```

Utilizando reducir

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                           arg2=s2,
                                           res=s1*s2))
    return s1 * s2

asequence = [1, 2, 3]
```

Dado un `initializer` la función se inicia aplicándola al inicializador y al primer elemento iterable:

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30
```

Sin el parámetro `initializer` , la `reduce` inicia al aplicar la función a los dos primeros elementos de la lista:

```
cumprod = reduce(multiply, asequence)
# Out: 1 * 2 = 2
```

```
#      2 * 3 = 6
print(cumprod)
# Out: 6
```

Producto acumulativo

```
import operator
reduce(operator.mul, [10, 5, -3])
# Out: -150
```

Variante sin cortocircuito de alguno / todos.

`reduce` no terminará la iteración antes de que el `iterable` se haya iterado completamente, por lo que se puede usar para crear una función sin cortocircuito en `any()` o `all()` :

```
import operator
# non short-circuit "all"
reduce(operator.and_, [False, True, True, True]) # = False

# non short-circuit "any"
reduce(operator.or_, [True, False, False, False]) # = True
```

Primer elemento verdadero / falso de una secuencia (o último elemento si no hay ninguno)

```
# First falsy element or last element if all are truthy:
reduce(lambda i, j: i and j, [100, [], 20, 10])    # = []
reduce(lambda i, j: i and j, [100, 50, 20, 10])    # = 10

# First truthy element or last element if all falsy:
reduce(lambda i, j: i or j, [100, [], 20, 0])     # = 100
reduce(lambda i, j: i or j, ['', {}, [], None])   # = None
```

En lugar de crear una función `lambda` , generalmente se recomienda crear una función nombrada:

```
def do_or(i, j):
    return i or j

def do_and(i, j):
    return i and j

reduce(do_or, [100, [], 20, 0])                   # = 100
reduce(do_and, [100, [], 20, 0])                  # = []
```

Lea Reducir en línea: https://riptutorial.com/es/python/topic/328/reducir

Capítulo 177: Representaciones de cadena de instancias de clase: métodos __str__ y __repr__

Observaciones

Una nota sobre la implementación de ambos métodos.

Cuando se implementan ambos métodos, es algo común tener un método `__str__` que devuelve una representación amigable para el ser humano (por ejemplo, "As of Spaces") y `__repr__` devuelve una representación amigable de `eval` .

De hecho, la documentación de Python para `repr()` nota exactamente esto:

> Para muchos tipos, esta función intenta devolver una cadena que produciría un objeto con el mismo valor cuando se pasa a eval (), de lo contrario, la representación es una cadena entre paréntesis angulares que contiene el nombre del tipo de objeto. con información adicional que a menudo incluye el nombre y la dirección del objeto.

Lo que eso significa es que `__str__` podría implementarse para devolver algo como "As of Spaces" como se mostró anteriormente, `__repr__` podría implementarse para devolver una `Card('Spades', 1)`

Esta cadena podría pasarse directamente a `eval` en una especie de "ida y vuelta":

```
object -> string -> object
```

Un ejemplo de una implementación de tal método podría ser:

```
def __repr__(self):
    return "Card(%s, %d)" % (self.suit, self.pips)
```

Notas

[1] Esta salida es específica de la implementación. La cadena mostrada es de cpython.

[2] Es posible que ya hayas visto el resultado de esta división `str()` / `repr()` y no lo hayas conocido. Cuando las cadenas que contienen caracteres especiales, como las barras invertidas, se convierten en cadenas a través de `str()` las barras diagonales aparecen como están (aparecen una vez). Cuando se convierten en cadenas mediante `repr()` (por ejemplo, como

elementos de una lista que se muestra), las barras invertidas se escapan y, por lo tanto, aparecen dos veces.

Examples

Motivación

Así que acabas de crear tu primera clase en Python, una clase pequeña y ordenada que encapsula una carta de juego:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips
```

En otra parte de tu código, creas algunas instancias de esta clase:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)
```

Incluso has creado una lista de cartas para representar una "mano":

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

Ahora, durante la depuración, quieres ver cómo se ve tu mano, así que haces lo que es natural y escribes:

```
print(my_hand)
```

Pero lo que obtienes es un montón de galimatías:

```
[<__main__.Card instance at 0x0000000002533788>,
 <__main__.Card instance at 0x00000000025B95C8>,
 <__main__.Card instance at 0x00000000025FF508>]
```

Confundido, intenta simplemente imprimir una sola tarjeta:

```
print(ace_of_spades)
```

Y de nuevo, obtienes esta salida extraña:

```
<__main__.Card instance at 0x0000000002533788>
```

No tener miedo. Estamos a punto de arreglar esto.

Primero, sin embargo, es importante entender lo que está pasando aquí. Cuando escribió `print(ace_of_spades)` le dijo a Python que quería que imprimiera información sobre la instancia de

la `Card` su código está llamando `ace_of_spades` . Y para ser justos, lo hizo.

Esa salida se compone de dos bits importantes: el `type` de objeto y la `id` del objeto. La segunda parte sola (el número hexadecimal) es suficiente para identificar de forma única el objeto en el momento de la llamada de `print` . [1]

Lo que realmente sucedió fue que le pediste a Python que "expresara con palabras" la esencia de ese objeto y luego te lo mostrara. Una versión más explícita de la misma maquinaria podría ser:

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```

En la primera línea, intenta convertir la instancia de su `Card` en una cadena, y en la segunda, la muestra.

El problema

El problema con el que se encuentra surge debido al hecho de que, mientras le contaba a Python todo lo que necesitaba saber sobre la clase de la `Card` para que usted *creara las* tarjetas, *no le* dijo cómo quería que las instancias de la `Card` se convirtieran en cadenas.

Y como no lo sabía, cuando usted (implícitamente) escribió `str(ace_of_spades)` , le dio lo que vio, una representación genérica de la instancia de la `Card` .

La Solución (Parte 1)

Pero *podemos* decirle a Python cómo queremos que las instancias de nuestras clases personalizadas se conviertan en cadenas. Y la forma en que lo hacemos es con el método `__str__` "dunder" (para subrayado doble) o "magic".

Siempre que le digas a Python que cree una cadena desde una instancia de clase, buscará un método `__str__` en la clase y lo llamará.

Considere la siguiente versión actualizada de nuestra clase de `Card` :

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

        card_name = special_names.get(self.pips, str(self.pips))

        return "%s of %s" % (card_name, self.suit)
```

Aquí, ahora hemos definido el método `__str__` en nuestra clase de `Card` que, después de una

simple búsqueda en el diccionario de tarjetas de caras, **devuelve** una cadena con el formato que decidamos.

(Tenga en cuenta que las "devoluciones" están en negrita aquí, para resaltar la importancia de devolver una cadena y no simplemente de imprimirla. La impresión puede parecer que funciona, pero luego se imprimirá la tarjeta cuando hizo algo como `str(ace_of_spades)` , sin siquiera tener una llamada a la función de impresión en su programa principal. Para que quede claro, asegúrese de que `__str__` devuelva una cadena.

El método `__str__` es un método, por lo que el primer argumento será `self` y no debería aceptar, ni pasar argumentos adicionales.

Volviendo a nuestro problema de mostrar la tarjeta de una manera más fácil para el usuario, si volvemos a ejecutar:

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

Veremos que nuestra salida es mucho mejor:

```
Ace of Spades
```

Muy bien, hemos terminado, ¿verdad?

Bueno, solo para cubrir nuestras bases, verifiquemos que hemos resuelto el primer problema que encontramos, imprimiendo la lista de instancias de la `Card` , la `hand` .

Así que volvemos a comprobar el siguiente código:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

Y, para nuestra sorpresa, volvemos a obtener esos divertidos códigos hex:

```
[<__main__.Card instance at 0x00000000026F95C8>,
 <__main__.Card instance at 0x000000000273F4C8>,
 <__main__.Card instance at 0x0000000002732E08>]
```

¿Que esta pasando? Le dijimos a Python cómo queríamos que se mostraran nuestras instancias de la `Card` , ¿por qué parece que aparentemente se olvidó?

La Solución (Parte 2)

Bueno, la maquinaria detrás de escena es un poco diferente cuando Python quiere obtener la representación de cadena de los elementos en una lista. Resulta que a Python no le importa `__str__` para este propósito.

En su lugar, busca un método diferente, `__repr__` , y si *eso* no *es* encontrado, se recurre a la "cosa

hexadecimal". [2]

Entonces, ¿estás diciendo que tengo que hacer dos métodos para hacer lo mismo? ¿Uno para cuando quiero `print` mi tarjeta por sí mismo y otro cuando está en algún tipo de contenedor?

No, pero primero veamos cómo *sería* nuestra clase si implementáramos los métodos `__str__` y `__repr__` :

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (S)" % (card_name, self.suit)

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (R)" % (card_name, self.suit)
```

Aquí, la implementación de los dos métodos `__str__` y `__repr__` son exactamente iguales, excepto que, para diferenciar los dos métodos, se agrega `(S)` a las cadenas devueltas por `__str__` y `(R)` se agrega a las cadenas devueltas por `__repr__` .

Tenga en cuenta que al igual que nuestro método `__str__` , `__repr__` no acepta argumentos y **devuelve** una cadena.

Podemos ver ahora qué método es responsable de cada caso:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)

my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]

print(my_hand)           # [Ace of Spades (R), 4 of Clubs (R), 6 of Hearts (R)]

print(ace_of_spades)     # Ace of Spades (S)
```

Como se cubrió, se llamó al método `__str__` cuando pasamos nuestra instancia de `Card` a `print` y al método `__repr__` cuando pasamos *una lista de nuestras instancias* a `print` .

En este punto, vale la pena señalar que al igual que podemos crear explícitamente una cadena desde una instancia de clase personalizada usando `str()` como lo hicimos anteriormente, también podemos crear explícitamente una **representación** de **cadena** de nuestra clase con una función incorporada llamada `repr()` .

Por ejemplo:

```
str_card = str(four_of_clubs)
```

```
print(str_card)                      # 4 of Clubs (S)

repr_card = repr(four_of_clubs)
print(repr_card)                     # 4 of Clubs (R)
```

Y además, si está definido, *podríamos* llamar a los métodos directamente (aunque parece un poco incierto e innecesario):

```
print(four_of_clubs.__str__())     # 4 of Clubs (S)

print(four_of_clubs.__repr__())    # 4 of Clubs (R)
```

Sobre esas funciones duplicadas ...

Los desarrolladores de Python se dieron cuenta de que, en el caso de que quisieras que se devolvieran cadenas idénticas desde `str()` y `repr()` es posible que tengas que duplicar funcionalmente los métodos, algo que a nadie le gusta.

Entonces, en cambio, existe un mecanismo para eliminar la necesidad de eso. Una que te colgué hasta este punto. Resulta que si una clase implementa el método `__repr__` *pero no* el método `__str__` , y usted pasa una instancia de esa clase a `str()` (ya sea de manera implícita o explícita), Python `__repr__` su implementación `__repr__` y lo usará.

Entonces, para ser claros, considere la siguiente versión de la clase de `Card` :

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)
```

Tenga en cuenta que esta versión *solo* implementa el método `__repr__` . No obstante, las llamadas a `str()` dan como resultado una versión fácil de usar:

```
print(six_of_hearts)            # 6 of Hearts  (implicit conversion)
print(str(six_of_hearts))       # 6 of Hearts  (explicit conversion)
```

al igual que las llamadas a `repr()` :

```
print([six_of_hearts])          #[6 of Hearts] (implicit conversion)
print(repr(six_of_hearts))      # 6 of Hearts  (explicit conversion)
```

Resumen

Para que pueda habilitar las instancias de su clase para que se "muestren" de manera fácil de usar, querrá considerar implementar al menos el método `__repr__` su clase. Si la memoria sirve, durante una charla, Raymond Hettinger dijo que asegurarse de que las clases implementen `__repr__` es una de las primeras cosas que busca mientras hace revisiones de código Python, y hasta ahora debería estar claro por qué. La cantidad de información que *podría* haber agregado a las declaraciones de depuración, los informes de fallos o los archivos de registro con un método simple es abrumadora en comparación con la información más escasa y, a menudo, poco útil (tipo, id) que se proporciona de forma predeterminada.

Si desea *diferentes* representaciones para cuando, por ejemplo, dentro de un contenedor, querrá implementar los métodos `__repr__` y `__str__` . (Más sobre cómo puede usar estos dos métodos de manera diferente a continuación).

Ambos métodos implementados, eval-round-trip style __repr __ ()

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    # Called when instance is converted to a string via str()
    # Examples:
    #   print(card1)
    #   print(str(card1)
    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)

    # Called when instance is converted to a string via repr()
    # Examples:
    #   print([card1, card2, card3])
    #   print(repr(card1))
    def __repr__(self):
        return "Card(%s, %d)" % (self.suit, self.pips)
```

Lea Representaciones de cadena de instancias de clase: métodos __str__ y __repr__ en línea: https://riptutorial.com/es/python/topic/4845/representaciones-de-cadena-de-instancias-de-clase--metodos---str---y---repr--

Capítulo 178: Sangría

Examples

Errores de sangría

El espacio debe ser uniforme y uniforme en todo. La sangría incorrecta puede causar un `IndentationError` o hacer que el programa haga algo inesperado. El siguiente ejemplo genera un `IndentationError` :

```
a = 7
if a > 5:
  print "foo"
else:
  print "bar"
 print "done"
```

O si la línea que sigue a dos puntos no tiene sangría, también se levantará un `IndentationError` :

```
if True:
print "true"
```

Si agrega sangría donde no pertenece, se generará un `IndentationError` :

```
if  True:
    a = 6
        b = 5
```

Si olvida desmarcar la funcionalidad podría perderse. En este ejemplo, `None` se devuelve en lugar del `False` esperado:

```
def isEven(a):
    if a%2 ==0:
        return True
        #this next line should be even with the if
        return False
print isEven(7)
```

Ejemplo simple

Para Python, Guido van Rossum basó la agrupación de declaraciones en sangría. Las razones de esto se explican en la primera sección de las "Preguntas frecuentes sobre Python de diseño e historia" . Los dos puntos, `:` , se utilizan para declarar un bloque de código con sangría , como el siguiente ejemplo:

```
class ExampleClass:
    #Every function belonging to a class must be indented equally
    def __init__(self):
```

```
        name = "example"

    def someFunction(self, a):
        #Notice everything belonging to a function must be indented
        if a > 5:
            return True
        else:
            return False

#If a function is not indented to the same level it will not be considers as part of the
parent class
def separateFunction(b):
    for i in b:
    #Loops are also indented and nested conditions start a new indentation
        if i == 1:
            return True
    return False

separateFunction([2,3,5,6,1])
```

¿Espacios o pestañas?

La sangría recomendada es de 4 espacios, pero se pueden usar tabulaciones o espacios siempre que sean consistentes. ***No mezcle tabulaciones y espacios en Python*** ya que esto causará un error en Python 3 y puede causar errores en Python 2 .

Cómo se analiza la sangría

Los espacios en blanco son manejados por el analizador léxico antes de ser analizados.

El analizador léxico usa una pila para almacenar niveles de sangría. Al principio, la pila contiene solo el valor 0, que es la posición más a la izquierda. Cada vez que comienza un bloque anidado, el nuevo nivel de sangría se empuja en la pila, y se inserta un token "INDENT" en el flujo de token que se pasa al analizador. Nunca puede haber más de un token "INDENT" en una fila (`IndentationError`).

Cuando se encuentra una línea con un nivel de sangría más pequeño, los valores se extraen de la pila hasta que un valor está en la parte superior, que es igual al nuevo nivel de sangría (si no se encuentra ninguno, se produce un error de sintaxis). Para cada valor que aparece, se genera un token "DEDENT". Obviamente, puede haber múltiples tokens "DEDENT" en una fila.

El analizador léxico omite líneas vacías (aquellas que solo contienen espacios en blanco y posiblemente comentarios), y nunca generará tokens "INDENT" o "DEDENT" para ellas.

Al final del código fuente, se generan tokens "DEDENT" para cada nivel de sangrado que queda en la pila, hasta que solo queda el 0.

Por ejemplo:

```
if foo:
    if bar:
        x = 42
```

```
else:
    print foo
```

se analiza como:

```
<if> <foo> <:>                    [0]
<INDENT> <if> <bar> <:>           [0, 4]
<INDENT> <x> <=> <42>             [0, 4, 8]
<DEDENT> <DEDENT> <else> <:>      [0]
<INDENT> <print> <foo>            [0, 2]
<DEDENT>
```

El analizador que maneja los tokens "INDENT" y "DEDENT" como delimitadores de bloque.

Lea Sangría en línea: https://riptutorial.com/es/python/topic/2597/sangria

Capítulo 179: Seguridad y criptografía

Introducción

Python, siendo uno de los lenguajes más populares en seguridad de computadoras y redes, tiene un gran potencial en seguridad y criptografía. Este tema trata sobre las funciones criptográficas y las implementaciones en Python desde sus usos en seguridad informática y de red hasta algoritmos de hash y cifrado / descifrado.

Sintaxis

- hashlib.new (nombre)
- hashlib.pbkdf2_hmac (nombre, contraseña, salt, rounds, dklen = Ninguno)

Observaciones

Muchos de los métodos en `hashlib` requerirán que usted pase valores interpretables como buffers de bytes, en lugar de cadenas. Este es el caso de `hashlib.new().update()` así como `hashlib.pbkdf2_hmac` . Si tiene una cadena, puede convertirla en un búfer de bytes al añadir el carácter `b` al comienzo de la cadena:

```
  "This is a string"
 b"This is a buffer of bytes"
```

Examples

Cálculo de un resumen del mensaje

El módulo `hashlib` permite crear generadores de resumen de mensajes a través del `new` método. Estos generadores convertirán una cadena arbitraria en un compendio de longitud fija:

```
import hashlib

h = hashlib.new('sha256')
h.update(b'Nobody expects the Spanish Inquisition.')
h.digest()
# ==>
b'.\xdf\xda\xdaVR[\x12\x90\xff\x16\xfb\x17D\xcf\xb4\x82\xdd)\x14\xff\xbc\xb6Iy\x0c\x0eX\x9eF-
='
```

Tenga en cuenta que puede llamar a la `update` un número arbitrario de veces antes de llamar al `digest` que es útil para agrupar un fragmento de archivo grande por fragmento. También puede obtener el resumen en formato hexadecimal utilizando `hexdigest` :

```
h.hexdigest()
```

```
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d'
```

Algoritmos de hash disponibles

`hashlib.new` requiere el nombre de un algoritmo cuando lo llama para producir un generador. Para averiguar qué algoritmos están disponibles en el intérprete de Python actual, use `hashlib.algorithms_available` :

```
import hashlib
hashlib.algorithms_available
# ==> {'sha256', 'DSA-SHA', 'SHA512', 'SHA224', 'dsaWithSHA', 'SHA', 'RIPEMD160', 'ecdsa-with-
SHA1', 'sha1', 'SHA384', 'md5', 'SHA1', 'MD5', 'MD4', 'SHA256', 'sha384', 'md4', 'ripemd160',
'sha224', 'sha512', 'DSA', 'dsaEncryption', 'sha', 'whirlpool'}
```

La lista devuelta variará según la plataforma y el intérprete; Asegúrate de comprobar que tu algoritmo está disponible.

También hay algunos algoritmos que están *garantizados* para estar disponibles en todas las plataformas e intérpretes, que están disponibles usando `hashlib.algorithms_guaranteed` :

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

Contraseña segura Hashing

El algoritmo `hashlib` expuesto por el módulo `hashlib` se puede usar para realizar el hashing de contraseña seguro. Si bien este algoritmo no puede evitar los ataques de fuerza bruta para recuperar la contraseña original del hash almacenado, hace que dichos ataques sean muy costosos.

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 puede funcionar con cualquier algoritmo de resumen, el ejemplo anterior utiliza SHA256, que generalmente se recomienda. La sal aleatoria debe almacenarse junto con la contraseña hash, la necesitará nuevamente para comparar una contraseña ingresada con el hash almacenado. Es esencial que cada contraseña tenga una sal diferente. En cuanto al número de rondas, se recomienda establecerlo lo más alto posible para su aplicación .

Si desea que el resultado sea hexadecimal, puede usar el módulo `binascii` :

```
import binascii
hexhash = binascii.hexlify(hash)
```

Nota : si bien PBKDF2 no es malo, bcrypt y especialmente scrypt se consideran más fuertes contra los ataques de fuerza bruta. Tampoco es parte de la biblioteca estándar de Python en este

momento.

Hash de archivo

Un hash es una función que convierte una secuencia de longitud variable de bytes en una secuencia de longitud fija. Los archivos de hash pueden ser ventajosos por muchas razones. Los hash se pueden usar para verificar si dos archivos son idénticos o verificar que el contenido de un archivo no se haya dañado o cambiado.

Puedes usar `hashlib` para generar un hash para un archivo:

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

Para archivos más grandes, se puede usar un búfer de longitud fija:

```
import hashlib
SIZE = 65536
hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    buffer = f.read(SIZE)
    while len(buffer) > 0:
        hasher.update(buffer)
        buffer = f.read(SIZE)
print(hasher.hexdigest())
```

Cifrado simétrico utilizando pycrypto

La funcionalidad criptográfica incorporada de Python se limita actualmente al hashing. El cifrado requiere un módulo de terceros como pycrypto . Por ejemplo, proporciona el algoritmo AES que se considera el estado de la técnica para el cifrado simétrico. El siguiente código cifrará un mensaje dado usando una frase de contraseña:

```
import hashlib
import math
import os

from Crypto.Cipher import AES

IV_SIZE = 16    # 128 bit, fixed for the AES algorithm
KEY_SIZE = 32   # 256 bit meaning AES-256, can also be 128 or 192 bits
SALT_SIZE = 16  # This size is arbitrary

cleartext = b'Lorem ipsum'
password = b'highly secure encryption password'
salt = os.urandom(SALT_SIZE)
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
```

```
                                dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]

encrypted = salt + AES.new(key, AES.MODE_CFB, iv).encrypt(cleartext)
```

El algoritmo AES toma tres parámetros: clave de cifrado, vector de inicialización (IV) y el mensaje real a cifrar. Si tiene una clave AES generada aleatoriamente, puede usarla directamente y simplemente generar un vector de inicialización aleatorio. Sin embargo, una frase de contraseña no tiene el tamaño correcto, ni sería recomendable utilizarla directamente, ya que no es realmente aleatoria y, por lo tanto, tiene una entropía relativamente pequeña. En su lugar, usamos la implementación incorporada del algoritmo PBKDF2 para generar un vector de inicialización de 128 bits y una clave de cifrado de 256 bits a partir de la contraseña.

Tenga en cuenta la sal aleatoria que es importante tener un vector de inicialización diferente y una clave para cada mensaje cifrado. Esto garantiza en particular que dos mensajes iguales no darán como resultado un texto cifrado idéntico, pero también evita que los atacantes reutilicen el trabajo empleado en adivinar una frase de contraseña en los mensajes cifrados con otra frase de contraseña. Esta sal debe almacenarse junto con el mensaje cifrado para poder derivar el mismo vector de inicialización y la clave para descifrar.

El siguiente código volverá a descifrar nuestro mensaje:

```
salt = encrypted[0:SALT_SIZE]
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                              dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]
cleartext = AES.new(key, AES.MODE_CFB, iv).decrypt(encrypted[SALT_SIZE:])
```

Generando firmas RSA usando pycrypto

RSA se puede utilizar para crear una firma de mensaje. Una firma válida solo se puede generar con acceso a la clave RSA privada, por lo que la validación es posible con la clave pública correspondiente. Por lo tanto, mientras la otra parte sepa su clave pública, podrán verificar el mensaje que usted firmará y no se modificará, por ejemplo, un enfoque utilizado para el correo electrónico. Actualmente, se requiere un módulo de terceros como pycrypto para esta funcionalidad.

```
import errno

from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5

message = b'This message is from me, I promise.'

try:
    with open('privkey.pem', 'r') as f:
        key = RSA.importKey(f.read())
except IOError as e:
    if e.errno != errno.ENOENT:
```

```
        raise
    # No private key, generate a new one. This can take a few seconds.
    key = RSA.generate(4096)
    with open('privkey.pem', 'wb') as f:
        f.write(key.exportKey('PEM'))
    with open('pubkey.pem', 'wb') as f:
        f.write(key.publickey().exportKey('PEM'))

hasher = SHA256.new(message)
signer = PKCS1_v1_5.new(key)
signature = signer.sign(hasher)
```

La verificación de la firma funciona de manera similar pero usa la clave pública en lugar de la clave privada:

```
with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
hasher = SHA256.new(message)
verifier = PKCS1_v1_5.new(key)
if verifier.verify(hasher, signature):
    print('Nice, the signature is valid!')
else:
    print('No, the message was signed with the wrong private key or modified')
```

Nota : los ejemplos anteriores utilizan el algoritmo de firma PKCS # 1 v1.5, que es muy común. pycrypto también implementa el nuevo algoritmo PKCS # 1 PSS, reemplazando `PKCS1_v1_5` por `PKCS1_PSS` en los ejemplos debería funcionar si desea usar ese. Sin embargo, actualmente parece que hay pocas razones para usarlo .

Cifrado RSA asimétrico utilizando pycrypto

El cifrado asimétrico tiene la ventaja de que un mensaje puede cifrarse sin intercambiar una clave secreta con el destinatario del mensaje. El remitente simplemente necesita conocer la clave pública de los destinatarios, esto permite cifrar el mensaje de manera que solo el destinatario designado (que tiene la clave privada correspondiente) pueda descifrarlo. Actualmente, se requiere un módulo de terceros como pycrypto para esta funcionalidad.

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message = b'This is a very secret message.'

with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
encrypted = cipher.encrypt(message)
```

El destinatario puede descifrar el mensaje si tiene la clave privada correcta:

```
with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
decrypted = cipher.decrypt(encrypted)
```

Nota : Los ejemplos anteriores utilizan el esquema de cifrado OPAP PKCS # 1. pycrypto también implementa el esquema de cifrado PKCS # 1 v1.5, este no se recomienda para nuevos protocolos, sin embargo, debido a advertencias conocidas .

Lea Seguridad y criptografía en línea: https://riptutorial.com/es/python/topic/2598/seguridad-y-criptografia

Capítulo 180: Serialización de datos

Sintaxis

- unpickled_string = pickle.loads (cadena)
- unpickled_string = pickle.load (file_object)
- pickled_string = pickle.dumps ([('', 'cmplx'), {('object',): None}], pickle.HIGHEST_PROTOCOL)
- pickle.dump (('', 'cmplx'), {('object',): None}], file_object, pickle.HIGHEST_PROTOCOL)
- unjsoned_string = json.loads (cadena)
- unjsoned_string = json.load (file_object)
- jsoned_string = json.dumps (('a', 'b', 'c', [1, 2, 3]))
- json.dump (('' a ',' b ',' c ', [1, 2, 3]), file_object)

Parámetros

Parámetro	Detalles
`protocol`	Usando `pickle` o `cPickle` , es el método por el cual los objetos se serializan o no serializan. Probablemente quieras usar `pickle.HIGHEST_PROTOCOL` aquí, lo que significa el método más nuevo.

Observaciones

¿Por qué usar JSON?

- Soporte de idiomas
- Legible para humanos
- A diferencia de Pickle, no tiene el peligro de ejecutar código arbitrario

¿Por qué no usar JSON?

- No admite tipos de datos Pythonic
- Las claves en los diccionarios no deben ser más que tipos de datos de cadena.

¿Por qué Pickle?

- Gran manera de serializar Pythonic (tuplas, funciones, clases)
- Las claves en los diccionarios pueden ser de cualquier tipo de datos.

¿Por qué no pickle?

- Falta soporte de idiomas
- No es seguro cargar datos arbitrarios.

Examples

Serialización utilizando JSON

JSON es un método de lenguaje cruzado, ampliamente utilizado para serializar datos.

Tipos de datos admitidos: *int* , *float* , *booleano* , *cadena* , *lista* y *dict* . Ver -> JSON Wiki para más

Aquí hay un ejemplo que demuestra el uso **básico** de **JSON** :

```
import json

families = (['John'], ['Mark', 'David', {'name': 'Avraham'}])

# Dumping it into string
json_families = json.dumps(families)
# [["John"], ["Mark", "David", {"name": "Avraham"}]]

# Dumping it to file
with open('families.json', 'w') as json_file:
    json.dump(families, json_file)

# Loading it from string
json_families = json.loads(json_families)

# Loading it from file
with open('families.json', 'r') as json_file:
    json_families = json.load(json_file)
```

Consulte el módulo JSON para obtener información detallada sobre JSON.

Serialización utilizando Pickle

Aquí hay un ejemplo que demuestra el uso **básico** de **pickle** :

```
# Importing pickle
try:
    import cPickle as pickle  # Python 2
except ImportError:
    import pickle  # Python 3

# Creating Pythonic object:
class Family(object):
    def __init__(self, names):
        self.sons = names

    def __str__(self):
        return ' '.join(self.sons)

my_family = Family(['John', 'David'])

# Dumping to string
pickle_data = pickle.dumps(my_family, pickle.HIGHEST_PROTOCOL)

# Dumping to file
```

```
with open('family.p', 'w') as pickle_file:
    pickle.dump(families, pickle_file, pickle.HIGHEST_PROTOCOL)

# Loading from string
my_family = pickle.loads(pickle_data)

# Loading from file
with open('family.p', 'r') as pickle_file:
    my_family = pickle.load(pickle_file)
```

Consulte Pickle para obtener información detallada acerca de Pickle.

ADVERTENCIA : La documentación oficial para pickle deja en claro que no hay garantías de seguridad. No cargues ningún dato en el que no confíes en su origen.

Lea Serialización de datos en línea: https://riptutorial.com/es/python/topic/3347/serializacion-de-datos

Capítulo 181: Serialización de datos de salmuera

Sintaxis

- pickle.dump (objeto, archivo, protocolo) # Para serializar un objeto
- pickle.load (archivo) # Para des-serializar un objeto
- pickle.dumps (objeto, protocolo) # Para serializar un objeto a bytes
- pickle.loads (buffer) # Para eliminar un serial de un objeto de bytes

Parámetros

Parámetro	Detalles
objeto	El objeto que se va a almacenar.
expediente	El archivo abierto que contendrá el objeto.
protocolo	El protocolo utilizado para el decapado del objeto (parámetro opcional)
buffer	Un objeto de bytes que contiene un objeto serializado.

Observaciones

Tipos pickleable

Los siguientes objetos son desmontables.

- `None` , `True` y `False`
- números (de todos los tipos)
- cuerdas (de todo tipo)
- `tuple` s, `list` s, `set` s y `dict` s que solo contienen objetos que se pueden recoger
- Funciones definidas en el nivel superior de un módulo.
- funciones integradas
- Clases que se definen en el nivel superior de un módulo.
    - instancias de dichas clases cuyo `__dict__` o el resultado de llamar a `__getstate__()` es seleccionable (consulte los documentos oficiales para obtener más información).

Basado en la documentación oficial de Python .

`pickle` y seguridad

El módulo pickle **no** es **seguro** . No debe utilizarse cuando se reciben datos serializados de una parte que no es de confianza, como a través de Internet.

Examples

Usando Pickle para serializar y deserializar un objeto

El módulo `pickle` implementa un algoritmo para convertir un objeto Python arbitrario en una serie de bytes. Este proceso también se llama **serializar** el objeto. El flujo de bytes que representa el objeto se puede transmitir o almacenar, y luego reconstruir para crear un nuevo objeto con las mismas características.

Para el código más simple, usamos las funciones `dump()` y `load()` .

Para serializar el objeto.

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

Deserializar el objeto.

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

Usando objetos de pickle y byte

También es posible serializar y deserializar objetos de byte, utilizando la función de `dumps` y `loads` ,

que son equivalentes a `dump` y `load` .

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) is bytes

deserialized_data = pickle.loads(serialized_data)
# deserialized_data == data
```

Personalizar datos en escabeche

Algunos datos no pueden ser decapados. Otros datos no deben ser decapados por otras razones.

Lo que será decapado se puede definir en el método `__getstate__` . Este método debe devolver algo que sea pickable.

En el lado opuesto está `__setstate__` : recibirá lo que `__getstate__` creó y tiene que inicializar el objeto.

```
class A(object):
    def __init__(self, important_data):
        self.important_data = important_data

        # Add data which cannot be pickled:
        self.func = lambda: 7

        # Add data which should never be pickled, because it expires quickly:
        self.is_up_to_date = False

    def __getstate__(self):
        return [self.important_data] # only this is needed

    def __setstate__(self, state):
        self.important_data = state[0]

        self.func = lambda: 7  # just some hard-coded unpicklable function

        self.is_up_to_date = False  # even if it was before pickling
```

Ahora, esto se puede hacer:

```
>>> a1 = A('very important')
>>>
>>> s = pickle.dumps(a1)  # calls a1.__getstate__()
>>>
>>> a2 = pickle.loads(s)  # calls a1.__setstate__(['very important'])
>>> a2
<__main__.A object at 0x0000000002742470>
>>> a2.important_data
'very important'
>>> a2.func()
7
```

La implementación aquí muestra una lista con un valor: `[self.important_data]` . Eso fue solo un ejemplo, `__getstate__` podría haber devuelto cualquier cosa que se pueda recoger, siempre y cuando `__setstate__` sepa cómo hacer el oppoisite. Una buena alternativa es un diccionario de

todos los valores: `{'important_data': self.important_data}` .

¡El constructor no se llama! Tenga en cuenta que en el ejemplo anterior, la instancia `a2` se creó en `pickle.loads` sin haber llamado a `A.__init__` , por lo que `A.__setstate__` tuvo que inicializar todo lo que `__init__` se habría inicializado si se hubiera llamado.

Lea Serialización de datos de salmuera en línea:
https://riptutorial.com/es/python/topic/2606/serializacion-de-datos-de-salmuera

Capítulo 182: Servidor HTTP de Python

Examples

Ejecutando un servidor HTTP simple

Python 2.x 2.3

```
python -m SimpleHTTPServer 9000
```

Python 3.x 3.0

```
python -m http.server 9000
```

La ejecución de este comando sirve los archivos del directorio actual en el puerto `9000` .

Si no se proporciona ningún argumento como número de puerto, el servidor se ejecutará en el puerto predeterminado `8000` .

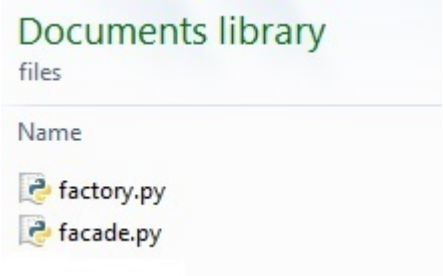
El indicador `-m` buscará en `sys.path` el archivo `.py` correspondiente para ejecutarse como un módulo.

Si solo quieres servir en localhost, deberás escribir un programa Python personalizado como:

```
import sys
import BaseHTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

HandlerClass = SimpleHTTPRequestHandler
ServerClass  = BaseHTTPServer.HTTPServer
Protocol     = "HTTP/1.0"

if sys.argv[1:]:
   port = int(sys.argv[1])
else:
   port = 8000
server_address = ('127.0.0.1', port)

HandlerClass.protocol_version = Protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

Archivos de servicio

Suponiendo que tiene el siguiente directorio de archivos:



Puede configurar un servidor web para servir estos archivos de la siguiente manera:

Python 2.x 2.3

```
import SimpleHTTPServer
import SocketServer

PORT = 8000

handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer(("localhost", PORT), handler)
print "Serving files at port {}".format(PORT)
httpd.serve_forever()
```

Python 3.x 3.0

```
import http.server
import socketserver

PORT = 8000

handler = http.server.SimpleHTTPRequestHandler
httpd = socketserver.TCPServer(("", PORT), handler)
print("serving at port", PORT)
httpd.serve_forever()
```

El módulo `SocketServer` proporciona las clases y funcionalidades para configurar un servidor de red.

`SocketServer` 's `TCPServer` clase configura un servidor utilizando el protocolo TCP. El constructor acepta una tupla que representa la dirección del servidor (es decir, la dirección IP y el puerto) y la clase que maneja las solicitudes del servidor.

El `SimpleHTTPRequestHandler` clase del `SimpleHTTPServer` módulo permite a los archivos en el directorio actual para ser servido.

Guarde el script en el mismo directorio y ejecútelo.

Ejecuta el servidor HTTP:

Python 2.x 2.3

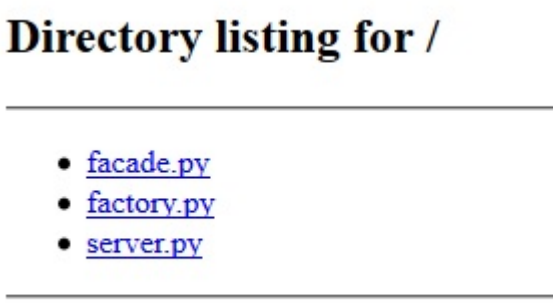
python -m SimpleHTTPServer 8000

Python 3.x 3.0

python -m http.server 8000

El indicador '-m' buscará 'sys.path' para que el archivo '.py' correspondiente se ejecute como un módulo.

Abra localhost: 8000 en el navegador, le dará lo siguiente:



API programática de SimpleHTTPServer

¿Qué sucede cuando ejecutamos `python -m SimpleHTTPServer 9000` ?

Para responder a esta pregunta, debemos entender el constructo de SimpleHTTPServer (https://hg.python.org/cpython/file/2.7/Lib/SimpleHTTPServer.py) y BaseHTTPServer (https://hg.python.org/cpython/file /2.7/Lib/BaseHTTPServer.py) .

En primer lugar, Python invoca el módulo `SimpleHTTPServer` con `9000` como argumento. Ahora observando el código SimpleHTTPServer,

```
def test(HandlerClass = SimpleHTTPRequestHandler,
         ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)


if __name__ == '__main__':
    test()
```

La función de prueba se invoca después de los controladores de solicitudes y ServerClass. Ahora se invoca BaseHTTPServer.test

```
def test(HandlerClass = BaseHTTPRequestHandler,
         ServerClass = HTTPServer, protocol="HTTP/1.0"):
"""Test the HTTP request handler class.

This runs an HTTP server on port 8000 (or the first command line
argument).

"""

if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000
```

```
server_address = ('', port)

HandlerClass.protocol_version = protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

Por lo tanto, aquí se analiza el número de puerto, que el usuario pasó como argumento y está vinculado a la dirección del host. Se llevan a cabo otros pasos básicos de programación de socket con puerto y protocolo dados. Finalmente se inicia el servidor socket.

Esta es una descripción básica de la herencia de la clase SocketServer a otras clases:

```
    +------------+
    | BaseServer |
    +------------+
          |
          v
    +-----------+        +------------------+
    | TCPServer |------->| UnixStreamServer |
    +-----------+        +------------------+
          |
          v
    +-----------+        +--------------------+
    | UDPServer |------->| UnixDatagramServer |
    +-----------+        +--------------------+
```

Los enlaces https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py y https://hg.python.org/cpython/file/2.7/Lib/SocketServer.py son útiles para encontrar más información. información.

Manejo básico de GET, POST, PUT usando BaseHTTPRequestHandler

```
# from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer # python2
from http.server import BaseHTTPRequestHandler, HTTPServer # python3
class HandleRequests(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        self._set_headers()
        self.wfile.write("received get request")

    def do_POST(self):
        '''Reads post request body'''
        self._set_headers()
        content_len = int(self.headers.getheader('content-length', 0))
        post_body = self.rfile.read(content_len)
        self.wfile.write("received post request:<br>{}".format(post_body))

    def do_PUT(self):
        self.do_POST()
```

```
host = ''
port = 80
HTTPServer((host, port), HandleRequests).serve_forever()
```

Ejemplo de salida utilizando `curl` :

```
$ curl http://localhost/
received get request%


$ curl -X POST http://localhost/
received post request:<br>%


$ curl -X PUT http://localhost/
received post request:<br>%


$ echo 'hello world' | curl --data-binary @- http://localhost/
received post request:<br>hello world
```

Lea Servidor HTTP de Python en línea: https://riptutorial.com/es/python/topic/4247/servidor-http-de-python

Capítulo 183: setup.py

Parámetros

Parámetro	Uso
`name`	Nombre de su distribución.
`version`	Cadena de versión de su distribución.
`packages`	Lista de paquetes de Python (es decir, directorios que contienen módulos) para incluir. Esto se puede especificar manualmente, pero en su `setuptools.find_packages()` se suele usar una llamada a `setuptools.find_packages()` .
`py_modules`	Lista de módulos de Python de nivel superior (es decir, archivos `.py` individuales) para incluir.

Observaciones

Para más información sobre el embalaje de python, consulte:

Introducción

Para escribir paquetes oficiales hay una guía de usuario de empaquetado .

Examples

Propósito de setup.py

El script de configuración es el centro de toda actividad en la construcción, distribución e instalación de módulos utilizando los Distutils. Su propósito es la correcta instalación del software.

Si todo lo que quiere hacer es distribuir un módulo llamado foo, contenido en un archivo foo.py, su secuencia de comandos de instalación puede ser tan simple como esto:

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Para crear una distribución de origen para este módulo, debe crear un script de configuración, setup.py, que contenga el código anterior, y ejecutar este comando desde una terminal:

```
python setup.py sdist
```

sdist creará un archivo de archivo (por ejemplo, tarball en Unix, archivo ZIP en Windows) que contiene su script de configuración setup.py y su módulo foo.py. El archivo comprimido se llamará foo-1.0.tar.gz (o .zip) y se descomprimirá en un directorio foo-1.0.

Si un usuario final desea instalar su módulo foo, todo lo que tiene que hacer es descargar foo-1.0.tar.gz (o .zip), descomprimirlo y, desde el directorio foo-1.0, ejecutar

```
python setup.py install
```

Agregando scripts de línea de comandos a su paquete de Python

Los guiones de línea de comando dentro de los paquetes de python son comunes. Puede organizar su paquete de tal manera que cuando un usuario instala el paquete, el script estará disponible en su ruta.

Si tenía el paquete de `greetings` que tenía la secuencia de comandos script `hello_world.py` .

```
greetings/
   greetings/
      __init__.py
      hello_world.py
```

Puedes ejecutar ese script ejecutando:

```
python greetings/greetings/hello_world.py
```

Sin embargo si quieres correrlo así:

```
hello_world.py
```

Puede lograr esto agregando `scripts` a su `setup()` en `setup.py` esta manera:

```
from setuptools import setup
setup(
  name='greetings',
  scripts=['hello_world.py']
)
```

Cuando instale el paquete de saludos ahora, se agregará a su ruta `hello_world.py` .

Otra posibilidad sería agregar un punto de entrada:

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

De esta manera solo tienes que ejecutarlo como:

```
greetings
```

Usando metadatos de control de fuente en setup.py

`setuptools_scm` es un paquete oficialmente bendecido que puede usar metadatos de Git o Mercurial para determinar el número de versión de su paquete, y encontrar paquetes de Python y datos de paquetes para incluir en ellos.

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

Este ejemplo utiliza ambas características; para usar solo los metadatos de SCM para la versión, reemplace la llamada a `find_packages()` con su lista de paquetes del manual, o para usar solo el buscador de paquetes, elimine `use_scm_version=True` .

Añadiendo opciones de instalación

Como se vio en ejemplos anteriores, el uso básico de este script es:

```
python setup.py install
```

Pero hay aún más opciones, como instalar el paquete y tener la posibilidad de cambiar el código y probarlo sin tener que volver a instalarlo. Esto se hace usando:

```
python setup.py develop
```

Si desea realizar acciones específicas como compilar una documentación de *Sphinx* o crear un código *fortran* , puede crear su propia opción como esta:

```
cmdclasses = dict()

class BuildSphinx(Command):

    """Build Sphinx documentation."""

    description = 'Build Sphinx documentation'
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        import sphinx
```

```
        sphinx.build_main(['setup.py', '-b', 'html', './doc', './doc/_build/html'])
        sphinx.build_main(['setup.py', '-b', 'man', './doc', './doc/_build/man'])

cmdclasses['build_sphinx'] = BuildSphinx

setup(
...
cmdclass=cmdclasses,
)
```

`initialize_options` y `finalize_options` se ejecutarán antes y después de la función de `run` , tal como lo sugieren sus nombres.

Después de eso, podrás llamar a tu opción:

```
python setup.py build_sphinx
```

Lea setup.py en línea: https://riptutorial.com/es/python/topic/1444/setup-py

Capítulo 184: Similitudes en la sintaxis, diferencias en el significado: Python vs. JavaScript

Introducción

A veces sucede que dos idiomas ponen significados diferentes en la misma expresión de sintaxis o similar. Cuando ambos lenguajes son de interés para un programador, aclarar estos puntos de bifurcación ayuda a comprender mejor los dos lenguajes en sus conceptos básicos y sutilezas.

Examples

`in` con listas

```
2 in [2, 3]
```

En Python esto se evalúa como Verdadero, pero en JavaScript como falso. Esto se debe a que en Python comprueba si un valor está contenido en una lista, por lo que 2 está en [2, 3] como su primer elemento. En JavaScript se usa con objetos y comprueba si un objeto contiene la propiedad con el nombre expresado por el valor. Así que JavaScript considera [2, 3] como un objeto o un mapa de clave-valor como este:

```
{'0': 2, '1': 3}
```

y verifica si tiene una propiedad o una clave '2' en ella. El entero 2 se convierte silenciosamente a la cadena '2'.

Lea Similitudes en la sintaxis, diferencias en el significado: Python vs. JavaScript en línea: https://riptutorial.com/es/python/topic/10766/similitudes-en-la-sintaxis--diferencias-en-el-significado--python-vs--javascript

Capítulo 185: Sobrecarga

Examples

Métodos de magia / Dunder

Los métodos de Magic (también llamados dunder como abreviatura de subrayado doble) en Python tienen un propósito similar al de la sobrecarga de operadores en otros idiomas. Permiten a una clase definir su comportamiento cuando se usa como un operando en expresiones de operador unarias o binarias. También sirven como implementaciones llamadas por algunas funciones integradas.

Considere esta implementación de vectores bidimensionales.

```
import math

class Vector(object):
    # instantiation
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # unary negation (-v)
    def __neg__(self):
        return Vector(-self.x, -self.y)

    # addition (v + u)
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # subtraction (v - u)
    def __sub__(self, other):
        return self + (-other)

    # equality (v == u)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # abs(v)
    def __abs__(self):
        return math.hypot(self.x, self.y)

    # str(v)
    def __str__(self):
        return '<{0.x}, {0.y}>'.format(self)

    # repr(v)
    def __repr__(self):
        return 'Vector({0.x}, {0.y})'.format(self)
```

Ahora es posible usar naturalmente instancias de la clase `Vector` en varias expresiones.

```
v = Vector(1, 4)
```

```
u = Vector(2, 0)

u + v           # Vector(3, 4)
print(u + v)    # "<3, 4>" (implicit string conversion)
u - v           # Vector(1, -4)
u == v          # False
u + v == v + u  # True
abs(u + v)      # 5.0
```

Contenedor y tipos de secuencia.

Es posible emular tipos de contenedores, que admiten el acceso a valores por clave o índice.

Considere esta implementación ingenua de una lista dispersa, que almacena solo los elementos que no son cero para conservar la memoria.

```
class sparselist(object):
    def __init__(self, size):
        self.size = size
        self.data = {}

    # l[index]
    def __getitem__(self, index):
        if index < 0:
            index += self.size
        if index >= self.size:
            raise IndexError(index)
        try:
            return self.data[index]
        except KeyError:
            return 0.0

    # l[index] = value
    def __setitem__(self, index, value):
        self.data[index] = value

    # del l[index]
    def __delitem__(self, index):
        if index in self.data:
            del self.data[index]

    # value in l
    def __contains__(self, value):
        return value == 0.0 or value in self.data.values()

    # len(l)
    def __len__(self):
        return self.size

    # for value in l: ...
    def __iter__(self):
        return (self[i] for i in range(self.size)) # use xrange for python2
```

Entonces, podemos usar una lista `sparselist` como una `list` normal.

```
l = sparselist(10 ** 6)  # list with 1 million elements
0 in l                   # True
```

```
10 in l                  # False

l[12345] = 10
10 in l                  # True
l[12345]                 # 10

for v in l:
    pass  # 0, 0, 0, ... 10, 0, 0 ... 0
```

Tipos callables

```
class adder(object):
    def __init__(self, first):
        self.first = first

    # a(...)
    def __call__(self, second):
        return self.first + second

add2 = adder(2)
add2(1)  # 3
add2(2)  # 4
```

Manejando conductas no implementadas.

Si su clase no implementa un operador sobrecargado específico para los tipos de argumentos provistos, debería `return NotImplemented` (**tenga en cuenta** que esta es una constante especial , no la misma que `NotImplementedError`). Esto permitirá a Python volver a probar otros métodos para hacer que la operación funcione:

> Cuando se devuelve `NotImplemented` , el intérprete intentará la operación reflejada en el otro tipo, o algún otro repliegue, dependiendo del operador. Si todas las operaciones intentadas devuelven No `NotImplemented` , el intérprete generará una excepción apropiada.

Por ejemplo, dado `x + y` , si `x.__add__(y)` devuelve no implementado, `y.__radd__(x)` se intenta en su lugar.

```
class NotAddable(object):

    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return NotImplemented


class Addable(NotAddable):

    def __add__(self, other):
        return Addable(self.value + other.value)

    __radd__ = __add__
```

Como este es el método *reflejado* , debemos implementar `__add__` **y** `__radd__` para obtener el comportamiento esperado en todos los casos; afortunadamente, como ambos están haciendo lo mismo en este simple ejemplo, podemos tomar un atajo.

En uso:

```
>>> x = NotAddable(1)
>>> y = Addable(2)
>>> x + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NotAddable' and 'NotAddable'
>>> y + y
<so.Addable object at 0x1095974d0>
>>> z = x + y
>>> z
<so.Addable object at 0x109597510>
>>> z.value
3
```

Sobrecarga del operador

A continuación se muestran los operadores que pueden sobrecargarse en clases, junto con las definiciones de métodos que se requieren y un ejemplo del operador en uso dentro de una expresión.

Nota: el uso de `other` como nombre de variable no es obligatorio, pero se considera la norma.

Operador	Método	Expresión
`+` Adición	`__add__(self, other)`	`a1 + a2`
`-` Resta	`__sub__(self, other)`	`a1 - a2`
`*` Multiplicación	`__mul__(self, other)`	`a1 * a2`
`@` Matrix Multiplication	`__matmul__(self, other)`	`a1 @ a2` (*Python 3.5*)
`/` División	`__div__(self, other)`	`a1 / a2` (*solo Python 2*)
`/` División	`__truediv__(self, other)`	`a1 / a2` (*Python 3*)
`//` División del piso	`__floordiv__(self, other)`	`a1 // a2`
`%` Modulo / resto	`__mod__(self, other)`	`a1 % a2`
`**` Poder	`__pow__(self, other[, modulo])`	`a1 ** a2`
`<<` Bitwise Left Shift	`__lshift__(self, other)`	`a1 << a2`
`>>` Bitwise Right Shift	`__rshift__(self, other)`	`a1 >> a2`

Operador	Método	Expresión
`&` Bitwise Y	`__and__(self, other)`	`a1 & a2`
`^` Bitwise XOR	`__xor__(self, other)`	`a1 ^ a2`
`|` (Bitwise OR)	`__or__(self, other)`	`a1 | a2`
`-` Negación (Aritmética)	`__neg__(self)`	`-a1`
`+` Positivo	`__pos__(self)`	`+a1`
`~` Bitwise NO	`__invert__(self)`	`~a1`
`<` Menos que	`__lt__(self, other)`	`a1 < a2`
`<=` Menor que o igual a	`__le__(self, other)`	`a1 <= a2`
`==` Igual a	`__eq__(self, other)`	`a1 == a2`
`!=` No es igual a	`__ne__(self, other)`	`a1 != a2`
`>` Mayor que	`__gt__(self, other)`	`a1 > a2`
`>=` Mayor que o igual a	`__ge__(self, other)`	`a1 >= a2`
`[index]` operador de índice	`__getitem__(self, index)`	`a1[index]`
`in` En operador	`__contains__(self, other)`	`a2 in a1`
`(*args, ...)` Llamando	`__call__(self, *args, **kwargs)`	`a1(*args, **kwargs)`

El `modulo` parámetro opcional para `__pow__` solo lo utiliza la función incorporada `pow` .

Cada uno de los métodos correspondientes a un operador *binario* tiene un método "correcto" correspondiente que comienza con `__r` , por ejemplo `__radd__` :

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, other):
        return self.a + other
    def __radd__(self, other):
        print("radd")
        return other + self.a

A(1) + 2  # Out:  3
2 + A(1)  # prints radd. Out: 3
```

así como una versión in situ correspondiente, comenzando con `__i` :

```
class B:
    def __init__(self, b):
        self.b = b
    def __iadd__(self, other):
        self.b += other
        print("iadd")
        return self

b = B(2)
b.b       # Out: 2
b += 1    # prints iadd
b.b       # Out: 3
```

Como no hay nada especial en estos métodos, muchas otras partes del lenguaje, partes de la biblioteca estándar e incluso módulos de terceros agregan métodos mágicos por sí mismos, como métodos para convertir un objeto en un tipo o verificar las propiedades del objeto. Por ejemplo, la función `str()` incorporada llama al método `__str__` del objeto, si existe. Algunos de estos usos se enumeran a continuación.

Función	Método	Expresión
Casting a `int`	`__int__(self)`	`int(a1)`
Función absoluta	`__abs__(self)`	`abs(a1)`
Casting a `str`	`__str__(self)`	`str(a1)`
Casting a `unicode`	`__unicode__(self)`	`unicode(a1)` (solo Python 2)
Representación de cuerdas	`__repr__(self)`	`repr(a1)`
Casting a `bool`	`__nonzero__(self)`	`bool(a1)`
Formato de cadena	`__format__(self, formatstr)`	`"Hi {:abc}".format(a1)`
Hash	`__hash__(self)`	`hash(a1)`
Longitud	`__len__(self)`	`len(a1)`
Invertido	`__reversed__(self)`	`reversed(a1)`
Piso	`__floor__(self)`	`math.floor(a1)`
Techo	`__ceil__(self)`	`math.ceil(a1)`

También existen los métodos especiales `__enter__` y `__exit__` para administradores de contexto, y muchos más.

Lea Sobrecarga en línea: https://riptutorial.com/es/python/topic/2063/sobrecarga

Capítulo 186: Sockets y cifrado / descifrado de mensajes entre el cliente y el servidor

Introducción

La criptografía se utiliza para fines de seguridad. No hay tantos ejemplos de cifrado / descifrado en Python utilizando el CTR de MODO de cifrado de IDEA. **Objetivo de esta documentación:**

Ampliación e implementación del esquema de firma digital RSA en la comunicación de estación a estación. Usando Hashing para la integridad del mensaje, eso es SHA-1. Produce un protocolo simple de transporte de llaves. Cifrar clave con cifrado IDEA. El modo de cifrado de bloque es el modo contador.

Observaciones

Idioma utilizado: Python 2.7 (enlace de descarga: https://www.python.org/downloads/)

Biblioteca utilizada:

* **PyCrypto** (Enlace de descarga: https://pypi.python.org/pypi/pycrypto)

* **PyCryptoPlus** (Enlace de descarga: https://github.com/doegox/python-cryptoplus)

Instalación de la biblioteca:

PyCrypto: descomprime el archivo. Vaya al directorio y abra el terminal para Linux (alt + ctrl + t) y CMD (Mayús + clic con el botón derecho + seleccionar indicador de comando abrir aquí) para Windows. Después de eso, escriba python setup.py install (Asegúrese de que Python Environment esté configurado correctamente en el sistema operativo Windows)

PyCryptoPlus: igual que la última biblioteca.

Implementación de tareas: La tarea se divide en dos partes. Uno es el proceso de apretón de manos y otro es el proceso de comunicación. Configuración de zócalo:

- Como la creación de claves públicas y privadas, así como el hashing de la clave pública, necesitamos configurar el socket ahora. Para configurar el socket, debemos importar otro módulo con "importar socket" y conectar (para el cliente) o vincular (para el servidor) la dirección IP y el puerto con el socket que recibe del usuario.

  ----------Lado del cliente----------

```
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
host = raw_input("Server Address To Be Connected -> ")
port = int(input("Port of The Server -> "))
server.connect((host, port))
```

----------Lado del servidor---------

```
try:
#setting up socket
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server.bind((host,port))
server.listen(5)
except BaseException: print "-----Check Server Address or Port-----"
```

"Socket.AF_INET, socket.SOCK_STREAM" nos permitirá utilizar la función accept () y los aspectos básicos de los mensajes. En lugar de eso, también podemos usar "socket.AF_INET, socket.SOCK_DGRAM", pero esa vez tendremos que usar setblocking (valor).

Proceso de apretón de manos:

- (CLIENTE) La primera tarea es crear claves públicas y privadas. Para crear la clave privada y pública, tenemos que importar algunos módulos. Son: desde Crypto import Random y desde Crypto.PublicKey import RSA. Para crear las claves, tenemos que escribir algunas líneas simples de códigos:

```
random_generator = Random.new().read
        key = RSA.generate(1024,random_generator)
        public = key.publickey().exportKey()
```

random_generator se deriva del módulo " **desde Crypto import Random** ". La clave se deriva de " **from Crypto.PublicKey import RSA** " que creará una clave privada, tamaño de 1024 generando caracteres aleatorios. Público está exportando clave pública desde clave privada generada anteriormente.

- (CLIENTE) Después de crear la clave pública y privada, debemos codificar la clave pública para enviarla al servidor mediante el hash SHA-1. Para usar el hash SHA-1 necesitamos importar otro módulo escribiendo "importar hashlib". Para codificar la clave pública, escribimos dos líneas de código:

```
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()
```

Aquí hash_object y hex_digest es nuestra variable. Después de esto, el cliente enviará hex_digest y public al servidor y el Servidor los verificará comparando el hash obtenido del cliente y el nuevo hash de la clave pública. Si el nuevo hash y el hash del cliente coinciden, pasará al siguiente procedimiento. Como el público enviado desde el cliente está en forma de cadena, no podrá utilizarse como clave en el lado del servidor. Para evitar esto y convertir la clave pública de cadena en rsa clave pública, necesitamos escribir `server_public_key = RSA.importKey(getpbk)` , aquí getpbk es la clave pública del cliente.

- (SERVIDOR) El siguiente paso es crear una clave de sesión. Aquí, he usado el módulo "os" para crear una clave aleatoria "key = os.urandom (16)" que nos dará una clave de 16 bits y después de eso he cifrado esa clave en "AES.MODE_CTR" y la hash de nuevo con SHA-1:

```
 #encrypt CTR MODE session key
 en = AES.new(key_128,AES.MODE_CTR,counter = lambda:key_128) encrypto =
en.encrypt(key_128)
 #hashing sha1
 en_object = hashlib.sha1(encrypto)
 en_digest = en_object.hexdigest()
```

Así que el en_digest será nuestra clave de sesión.

- (SERVIDOR) Para la parte final del proceso de protocolo de enlace es cifrar la clave pública obtenida del cliente y la clave de sesión creada en el lado del servidor.

```
 #encrypting session key and public key
 E = server_public_key.encrypt(encrypto,16)
```

Después del cifrado, el servidor enviará la clave al cliente como una cadena.

- (CLIENTE) Después de obtener la cadena cifrada (pública y clave de sesión) del servidor, el cliente los descifra usando la clave privada que se creó anteriormente junto con la clave pública. Como el cifrado (público y clave de sesión) estaba en forma de cadena, ahora debemos recuperarlo como clave mediante el uso de eval (). Si se realiza el descifrado, el proceso de intercambio se completa también, ya que ambas partes confirman que están utilizando las mismas claves. Para descifrar:

```
 en = eval(msg)
 decrypt = key.decrypt(en)
 # hashing sha1
 en_object = hashlib.sha1(decrypt) en_digest = en_object.hexdigest()
```

He usado el SHA-1 aquí para que sea legible en la salida.

Proceso de comunicación:

Para el proceso de comunicación, tenemos que usar la clave de sesión de ambos lados como la CLAVE para el cifrado IDEA MODE_CTR. Ambos lados cifrarán y descifrarán mensajes con IDEA.MODE_CTR usando la clave de sesión.

- (Cifrado) Para el cifrado IDEA, necesitamos una clave de 16 bits de tamaño y contador, como debe ser reclamable. El contador es obligatorio en MODE_CTR. La clave de la sesión que ciframos y hash tiene ahora un tamaño de 40 que excederá la clave límite del cifrado IDEA. Por lo tanto, necesitamos reducir el tamaño de la clave de sesión. Para reducir, podemos usar la función normal python integrada en la cadena de función [valor: valor]. Donde el valor puede ser cualquier valor según la elección del usuario. En nuestro caso, he hecho "clave [: 16]" donde tomará de 0 a 16 valores de la clave. Esta conversión se puede hacer de muchas maneras, como la clave [1:17] o la clave [16:]. La siguiente parte es crear una nueva función de cifrado de IDEA escribiendo IDEA.new () que tomará 3 argumentos para su procesamiento. El primer argumento será CLAVE, el segundo será el modo del cifrado de IDEA (en nuestro caso, IDEA.MODE_CTR) y el tercer argumento será el contador = que es una función que se debe llamar. El contador = mantendrá un tamaño de cadena

que será devuelto por la función. Para definir el contador =, debemos tener que usar unos valores razonables. En este caso, he usado el tamaño de la CLAVE definiendo lambda. En lugar de usar lambda, podríamos usar Counter.Util que genera un valor aleatorio para counter =. Para usar Counter.Util, necesitamos importar el módulo de contador desde crypto. Por lo tanto, el código será:

```
ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
```

Una vez que definimos el "IdeaEncrypt" como nuestra variable de cifrado IDEA, podemos usar la función integrada de cifrado para cifrar cualquier mensaje.

```
eMsg = ideaEncrypt.encrypt(whole)
#converting the encrypted message to HEXADECIMAL to readable eMsg =
eMsg.encode("hex").upper()
```

En este segmento de código, entero es el mensaje que se va a cifrar y eMsg es el mensaje cifrado. Después de cifrar el mensaje, lo he convertido en HEXADECIMAL para que sea legible y upper () es la función incorporada para hacer que los caracteres estén en mayúsculas. Después de eso, este mensaje cifrado se enviará a la estación opuesta para su descifrado.

- **(Descifrado)**

Para descifrar los mensajes cifrados, necesitaremos crear otra variable de cifrado utilizando los mismos argumentos y la misma clave, pero esta vez la variable descifrará los mensajes cifrados. El código para este mismo que la última vez. Sin embargo, antes de descifrar los mensajes, necesitamos decodificar el mensaje de hexadecimal porque en nuestra parte de cifrado, codificamos el mensaje cifrado en hexadecimal para que sea legible. Por lo tanto, todo el código será:

```
decoded = newmess.decode("hex")
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
dMsg = ideaDecrypt.decrypt(decoded)
```

Estos procesos se realizarán tanto en el servidor como en el lado del cliente para el cifrado y descifrado.

Examples

Implementación del lado del servidor

```
import socket
import hashlib
import os
import time
import itertools
import threading
import sys
import Crypto.Cipher.AES as AES
from Crypto.PublicKey import RSA
```

```
from CryptoPlus.Cipher import IDEA

#server address and port number input from admin
host= raw_input("Server Address - > ")
port = int(input("Port - > "))
#boolean for checking server and port
check = False
done = False

def animate():
    for c in itertools.cycle(['....','.......','..........','............']):
        if done:
            break
        sys.stdout.write('\rCHECKING IP ADDRESS AND NOT USED PORT '+c)
        sys.stdout.flush()
        time.sleep(0.1)
    sys.stdout.write('\r -----SERVER STARTED. WAITING FOR CLIENT-----\n')
try:
    #setting up socket
    server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    server.bind((host,port))
    server.listen(5)
    check = True
except BaseException:
    print "-----Check Server Address or Port-----"
    check = False

if check is True:
    # server Quit
    shutdown = False
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True
#binding client and address
client,address = server.accept()
print ("CLIENT IS CONNECTED. CLIENT'S ADDRESS ->",address)
print ("\n-----WAITING FOR PUBLIC KEY & PUBLIC KEY HASH-----\n")

#client's message(Public Key)
getpbk = client.recv(2048)

#conversion of string to KEY
server_public_key = RSA.importKey(getpbk)

#hashing the public key in server side for validating the hash from client
hash_object = hashlib.sha1(getpbk)
hex_digest = hash_object.hexdigest()

if getpbk != "":
    print (getpbk)
    client.send("YES")
    gethash = client.recv(1024)
    print ("\n-----HASH OF PUBLIC KEY----- \n"+gethash)
if hex_digest == gethash:
    # creating session key
    key_128 = os.urandom(16)
    #encrypt CTR MODE session key
    en = AES.new(key_128,AES.MODE_CTR,counter = lambda:key_128)
```

```
    encrypto = en.encrypt(key_128)
    #hashing sha1
    en_object = hashlib.sha1(encrypto)
    en_digest = en_object.hexdigest()

    print ("\n-----SESSION KEY-----\n"+en_digest)

    #encrypting session key and public key
    E = server_public_key.encrypt(encrypto,16)
    print ("\n-----ENCRYPTED PUBLIC KEY AND SESSION KEY-----\n"+str(E))
    print ("\n-----HANDSHAKE COMPLETE-----")
    client.send(str(E))
    while True:
        #message from client
        newmess = client.recv(1024)
        #decoding the message from HEXADECIMAL to decrypt the ecrypted version of the message
only
        decoded = newmess.decode("hex")
        #making en_digest(session_key) as the key
        key = en_digest[:16]
        print ("\nENCRYPTED MESSAGE FROM CLIENT -> "+newmess)
        #decrypting message from the client
        ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
        dMsg = ideaDecrypt.decrypt(decoded)
        print ("\n**New Message**  "+time.ctime(time.time()) +" > "+dMsg+"\n")
        mess = raw_input("\nMessage To Client -> ")
        if mess != "":
            ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
            eMsg = ideaEncrypt.encrypt(mess)
            eMsg = eMsg.encode("hex").upper()
            if eMsg != "":
                print ("ENCRYPTED MESSAGE TO CLIENT-> " + eMsg)
            client.send(eMsg)
    client.close()
else:
    print ("\n-----PUBLIC KEY HASH DOESNOT MATCH-----\n")
```

Implementación del lado del cliente

```
import time
import socket
import threading
import hashlib
import itertools
import sys
from Crypto import Random
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#animating loading
done = False
def animate():
    for c in itertools.cycle(['....','.......','..........','............']):
        if done:
            break
        sys.stdout.write('\rCONFIRMING CONNECTION TO SERVER '+c)
        sys.stdout.flush()
        time.sleep(0.1)
```

```
#public key and private key
random_generator = Random.new().read
key = RSA.generate(1024,random_generator)
public = key.publickey().exportKey()
private = key.exportKey()

#hashing the public key
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()

#Setting up socket
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

#host and port input user
host = raw_input("Server Address To Be Connected -> ")
port = int(input("Port of The Server -> "))
#binding the address and port
server.connect((host, port))
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True

def send(t,name,key):
    mess = raw_input(name + " : ")
    key = key[:16]
    #merging the message and the name
    whole = name+" : "+mess
    ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
    eMsg = ideaEncrypt.encrypt(whole)
    #converting the encrypted message to HEXADECIMAL to readable
    eMsg = eMsg.encode("hex").upper()
    if eMsg != "":
        print ("ENCRYPTED MESSAGE TO SERVER-> "+eMsg)
    server.send(eMsg)
def recv(t,key):
    newmess = server.recv(1024)
    print ("\nENCRYPTED MESSAGE FROM SERVER-> " + newmess)
    key = key[:16]
    decoded = newmess.decode("hex")
    ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
    dMsg = ideaDecrypt.decrypt(decoded)
    print ("\n**New Message From Server**  " + time.ctime(time.time()) + " : " + dMsg + "\n")

while True:
    server.send(public)
    confirm = server.recv(1024)
    if confirm == "YES":
        server.send(hex_digest)

    #connected msg
    msg = server.recv(1024)
    en = eval(msg)
    decrypt = key.decrypt(en)
    # hashing sha1
    en_object = hashlib.sha1(decrypt)
    en_digest = en_object.hexdigest()

    print ("\n-----ENCRYPTED PUBLIC KEY AND SESSION KEY FROM SERVER-----")
```

```
    print (msg)
    print ("\n-----DECRYPTED SESSION KEY-----")
    print (en_digest)
    print ("\n-----HANDSHAKE COMPLETE-----\n")
    alais = raw_input("\nYour Name -> ")

    while True:
        thread_send = threading.Thread(target=send,args=("------Sending Message------
",alais,en_digest))
        thread_recv = threading.Thread(target=recv,args=("------Recieving Message------
",en_digest))
        thread_send.start()
        thread_recv.start()

        thread_send.join()
        thread_recv.join()
        time.sleep(0.5)
    time.sleep(60)
    server.close()
```

Lea Sockets y cifrado / descifrado de mensajes entre el cliente y el servidor en línea: https://riptutorial.com/es/python/topic/8710/sockets-y-cifrado---descifrado-de-mensajes-entre-el-cliente-y-el-servidor

Capítulo 187: Subcomandos CLI con salida de ayuda precisa

Introducción

Diferentes formas de crear subcomandos como en `hg` o `svn` con la interfaz de línea de comandos exacta y la salida de ayuda, como se muestra en la sección Comentarios.

Análisis de los argumentos de la línea de comandos cubre un tema más amplio de análisis de argumentos.

Observaciones

Diferentes formas de crear subcomandos como en `hg` o `svn` con la interfaz de línea de comandos que se muestra en el mensaje de ayuda:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Examples

Forma nativa (sin bibliotecas)

```
"""
usage: sub <command>

commands:

  status -  show status
  list   -  print list
"""

import sys

def check():
    print("status")
    return 0

if sys.argv[1:] == ['status']:
    sys.exit(check())
elif sys.argv[1:] == ['list']:
    print("list")
else:
    print(__doc__.strip())
```

Salida sin argumentos:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Pros:

- no deps
- todo el mundo debería ser capaz de leer eso
- Control completo sobre el formato de ayuda.

argparse (formateador de ayuda predeterminado)

```
import argparse
import sys

def check():
    print("status")
    return 0

parser = argparse.ArgumentParser(prog="sub", add_help=False)
subparser = parser.add_subparsers(dest="cmd")

subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())
```

Salida sin argumentos:

```
usage: sub {status,list} ...

positional arguments:
  {status,list}
    status       show status
    list         print list
```

Pros:

- viene con Python
- Opción de análisis está incluido

argparse (formateador de ayuda personalizado)

Versión extendida de http://www.riptutorial.com/python/example/25282/argparse--default-help-formatter- que solucionó la salida de ayuda.

```
import argparse
import sys

class CustomHelpFormatter(argparse.HelpFormatter):
    def _format_action(self, action):
        if type(action) == argparse._SubParsersAction:
            # inject new class variable for subcommand formatting
            subactions = action._get_subactions()
            invocations = [self._format_action_invocation(a) for a in subactions]
            self._subcommand_max_length = max(len(i) for i in invocations)

        if type(action) == argparse._SubParsersAction._ChoicesPseudoAction:
            # format subcommand help line
            subcommand = self._format_action_invocation(action) # type: str
            width = self._subcommand_max_length
            help_text = ""
            if action.help:
                help_text = self._expand_help(action)
            return "  {:{width}} -  {}\n".format(subcommand, help_text, width=width)

        elif type(action) == argparse._SubParsersAction:
            # process subcommand help section
            msg = '\n'
            for subaction in action._get_subactions():
                msg += self._format_action(subaction)
            return msg
        else:
            return super(CustomHelpFormatter, self)._format_action(action)


def check():
    print("status")
    return 0

parser = argparse.ArgumentParser(usage="sub <command>", add_help=False,
             formatter_class=CustomHelpFormatter)

subparser = parser.add_subparsers(dest="cmd")
subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# custom help messge
parser._positionals.title = "commands"

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
```

```
    sys.exit(check())
```

Salida sin argumentos:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Lea Subcomandos CLI con salida de ayuda precisa en línea:
https://riptutorial.com/es/python/topic/7701/subcomandos-cli-con-salida-de-ayuda-precisa

Capítulo 188: sys

Introducción

El módulo **sys** proporciona acceso a funciones y valores relacionados con el entorno de ejecución del programa, como los parámetros de la línea de comando en `sys.argv` o la función `sys.exit()` para finalizar el proceso actual desde cualquier punto del flujo del programa.

Aunque está bien separado en un módulo, en realidad está incorporado y, como tal, siempre estará disponible en circunstancias normales.

Sintaxis

- Importe el módulo sys y haga que esté disponible en el espacio de nombres actual:

```
import sys
```

- Importe una función específica del módulo sys directamente al espacio de nombres actual:

```
from sys import exit
```

Observaciones

Para obtener detalles sobre todos los miembros del módulo **sys** , consulte la documentación oficial .

Examples

Argumentos de línea de comando

```
if len(sys.argv) != 4:         # The script name needs to be accounted for as well.
    raise RuntimeError("expected 3 command line arguments")

f = open(sys.argv[1], 'rb')    # Use first command line argument.
start_line = int(sys.argv[2])  # All arguments come as strings, so need to be
end_line = int(sys.argv[3])    # converted explicitly if other types are required.
```

Tenga en cuenta que en programas más grandes y más pulidos, usaría módulos como hacer clic para manejar los argumentos de la línea de comandos en lugar de hacerlo usted mismo.

Nombre del script

```
# The name of the executed script is at the beginning of the argv list.
print('usage:', sys.argv[0], '<filename> <start> <end>')
```

```
# You can use it to generate the path prefix of the executed program
# (as opposed to the current module) to access files relative to that,
# which would be good for assets of a game, for instance.
program_file = sys.argv[0]

import pathlib
program_path = pathlib.Path(program_file).resolve().parent
```

Flujo de error estándar

```
# Error messages should not go to standard output, if possible.
print('ERROR: We have no cheese at all.', file=sys.stderr)

try:
    f = open('nonexistent-file.xyz', 'rb')
except OSError as e:
    print(e, file=sys.stderr)
```

Finalización prematura del proceso y devolución de un código de salida.

```
def main():
    if len(sys.argv) != 4 or '--help' in sys.argv[1:]:
        print('usage: my_program <arg1> <arg2> <arg3>', file=sys.stderr)

        sys.exit(1)    # use an exit code to signal the program was unsuccessful

    process_data()
```

Lea sys en línea: https://riptutorial.com/es/python/topic/9847/sys

Capítulo 189: tempfile NamedTemporaryFile

Parámetros

param	descripción
modo	Modo para abrir archivo, por defecto = w + b
borrar	Para borrar el archivo al cierre, por defecto = Verdadero
sufijo	sufijo de nombre de archivo, por defecto = ''
prefijo	prefijo de nombre de archivo, por defecto = 'tmp'
dir	dirname para colocar tempfile, default = None
buffsize	por defecto = -1, (se usa por defecto el sistema operativo)

Examples

Cree (y escriba en) un archivo temporal persistente conocido

Puede crear archivos temporales que tengan un nombre visible en el sistema de archivos al que se puede acceder a través de la propiedad del `name` . El archivo se puede configurar en sistemas Unix para que se elimine en el cierre (establecido por `delete` param, el valor predeterminado es True) o se puede volver a abrir más tarde.

Lo siguiente creará y abrirá un archivo temporal nombrado y escribirá '¡Hola mundo!' a ese archivo. Se puede acceder a la `path` de archivo del archivo temporal a través del `name` , en este ejemplo, se guarda en la `path` variable y se imprime para el usuario. El archivo se vuelve a abrir después de cerrar el archivo y el contenido del archivo temporal se lee e imprime para el usuario.

```
import tempfile

with tempfile.NamedTemporaryFile(delete=False) as t:
    t.write('Hello World!')
    path = t.name
    print path

with open(path) as t:
    print t.read()
```

Salida:

```
/tmp/tmp6pireJ
Hello World!
```

Lea tempfile NamedTemporaryFile en línea: https://riptutorial.com/es/python/topic/3666/tempfile-namedtemporaryfile

Capítulo 190: Tipo de sugerencias

Sintaxis

- typing.Callable [[int, str], None] -> def func (a: int, b: str) -> None
- escribiendo.Mapping [str, int] -> {"a": 1, "b": 2, "c": 3}
- escribiendo.Lista [int] -> [1, 2, 3]
- escribiendo.configurar [int] -> {1, 2, 3}
- escribiendo.Opcional [int] -> Ninguno o int
- escribiendo.Secuencia [int] -> [1, 2, 3] o (1, 2, 3)
- escribiendo.cualquier -> cualquier tipo
- escribiendo.Union [int, str] -> 1 o "1"
- T = escribiendo.TypeVar ('T') -> Tipo genérico

Observaciones

La sugerencia de tipo, tal como se especifica en `PEP 484` , es una solución formalizada para indicar de forma estática el tipo de valor para el Código Python. Al aparecer junto al módulo de `typing` , las sugerencias de tipo ofrecen a los usuarios de Python la capacidad de anotar su código, ayudando así a los verificadores de tipos, mientras que, de forma indirecta, documentan su código con más información.

Examples

Tipos genéricos

El `typing.TypeVar` es una fábrica de tipo genérico. Su objetivo principal es servir como parámetro / marcador de posición para las anotaciones genéricas de función / clase / método:

```
import typing

T = typing.TypeVar("T")

def get_first_element(l: typing.Sequence[T]) -> T:
    """Gets the first element of a sequence."""
    return l[0]
```

Añadiendo tipos a una función

Tomemos un ejemplo de una función que recibe dos argumentos y devuelve un valor que indica su suma:

```
def two_sum(a, b):
    return a + b
```

Al observar este código, uno no puede indicar de forma segura y sin duda el tipo de argumentos para la función `two_sum` . Funciona tanto cuando se suministra con valores `int` :

```
print(two_sum(2, 1))  # result: 3
```

y con cuerdas:

```
print(two_sum("a", "b"))  # result: "ab"
```

y con otros valores, como `list` s, `tuple` s etcétera.

Debido a esta naturaleza dinámica de los tipos de python, donde muchos son aplicables para una operación determinada, cualquier verificador de tipo no podría afirmar razonablemente si una llamada para esta función debería permitirse o no.

Para ayudar a nuestro verificador de tipos, ahora podemos proporcionarle sugerencias de tipo en la definición de función que indica el tipo que permitimos.

Para indicar que solo queremos permitir tipos `int` , podemos cambiar nuestra definición de función para que se vea así:

```
def two_sum(a: int, b: int):
    return a + b
```

Las anotaciones siguen al nombre del argumento y están separadas por un carácter `:` .

De forma similar, para indicar que solo se permiten los tipos `str` , cambiaríamos nuestra función para especificarlo:

```
def two_sum(a: str, b: str):
    return a + b
```

Además de especificar el tipo de los argumentos, también se podría indicar el valor de retorno de una llamada de función. Esto se hace agregando el carácter `->` seguido del tipo después del paréntesis de cierre en la lista de argumentos *pero* antes de `:` al final de la declaración de la función:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

Ahora hemos indicado que el valor de retorno al llamar a `two_sum` debe ser de tipo `int` . De manera similar, podemos definir valores apropiados para `str` , `float` , `list` , `set` y otros.

Aunque las sugerencias de tipo son utilizadas principalmente por los verificadores de tipo y los IDE, a veces es posible que necesite recuperarlos. Esto se puede hacer usando el `__annotations__` especial `__annotations__` :

```
two_sum.__annotations__
```

```
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Miembros de la clase y métodos

```
class A:
    x = None  # type: float
    def __init__(self, x: float) -> None:
        """
        self should not be annotated
        init should be annotated to return None
        """
        self.x = x

    @classmethod
    def from_int(cls, x: int) -> 'A':
        """
        cls should not be annotated
        Use forward reference to refer to current class with string literal 'A'
        """
        return cls(float(x))
```

La referencia hacia adelante de la clase actual es necesaria ya que las anotaciones se evalúan cuando se define la función. Las referencias directas también se pueden usar cuando se hace referencia a una clase que causaría una importación circular si se importara.

Variables y atributos

Las variables son anotadas usando comentarios:

```
x = 3  # type: int
x = negate(x)
x = 'a type-checker might catch this error'
```

Python 3.x 3.6

A partir de Python 3.6, también hay una nueva sintaxis para las anotaciones de variables . El código de arriba podría usar el formulario

```
x: int = 3
```

A diferencia de los comentarios, también es posible simplemente agregar una sugerencia de tipo a una variable que no se declaró anteriormente, sin establecer un valor para ella:

```
y: int
```

Además, si se utilizan en el módulo o en el nivel de clase, las sugerencias de tipo se pueden recuperar utilizando `typing.get_type_hints(class_or_module)` :

```
class Foo:
    x: int
    y: str = 'abc'
```

```
print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

Alternativamente, se puede acceder a ellos usando el atributo o variable especial `__annotations__` :

```
x: int
print(__annotations__)
# {'x': <class 'int'>}

class C:
    s: str
print(C.__annotations__)
# {'s': <class 'str'>}
```

NamedTuple

La creación de una doble con nombre con sugerencias de tipo se realiza utilizando la función NamedTuple del módulo de `typing` :

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

Tenga en cuenta que el nombre del tipo resultante es el primer argumento de la función, pero debe asignarse a una variable con el mismo nombre para facilitar el trabajo de los verificadores de tipos.

Escriba sugerencias para argumentos de palabras clave

```
def hello_world(greeting: str = 'Hello'):
    print(greeting + ' world!')
```

Tenga en cuenta los espacios alrededor del signo igual en contraposición a la forma en que se suelen diseñar los argumentos de palabras clave.

Lea Tipo de sugerencias en línea: https://riptutorial.com/es/python/topic/1766/tipo-de-sugerencias

Capítulo 191: Tipos de datos de Python

Introducción

Los tipos de datos no son más que variables que utilizaste para reservar algo de espacio en la memoria. Las variables de Python no necesitan una declaración explícita para reservar espacio de memoria. La declaración ocurre automáticamente cuando asignas un valor a una variable.

Examples

Tipo de datos numeros

Los números tienen cuatro tipos en Python. Int, flotador, complejo, y largo.

```
int_num = 10    #int value
float_num = 10.2    #float value
complex_num = 3.14j    #complex value
long_num = 1234567L    #long value
```

Tipo de datos de cadena

Las cadenas se identifican como un conjunto contiguo de caracteres representados en las comillas. Python permite pares de comillas simples o dobles. Las cadenas son tipos de datos de secuencia inmutables, es decir, cada vez que se realizan cambios en una cadena, se crea un objeto de cadena completamente nuevo.

```
a_str = 'Hello World'
print(a_str)    #output will be whole string. Hello World
print(a_str[0])    #output will be first character. H
print(a_str[0:5])    #output will be first five characters. Hello
```

Tipo de datos de lista

Una lista contiene elementos separados por comas y encerrados entre corchetes []. Las listas son casi similares a las matrices en C. Una diferencia es que todos los elementos que pertenecen a una lista pueden ser de diferentes tipos de datos.

```
list = [123,'abcd',10.2,'d']    #can be a array of any data type or single data type.
list1 = ['hello','world']
print(list)    #will ouput whole list. [123,'abcd',10.2,'d']
print(list[0:2])    #will output first two element of list. [123,'abcd']
print(list1 * 2)    #will gave list1 two times. ['hello','world','hello','world']
print(list + list1)    #will gave concatenation of both the lists.
[123,'abcd',10.2,'d','hello','world']
```

Tipo de datos de la tupla

Las listas están entre paréntesis [] y sus elementos y tamaño pueden cambiarse, mientras que las tuplas están entre paréntesis () y no se pueden actualizar. Las tuplas son inmutables.

```
tuple = (123,'hello')
tuple1 = ('world')
print(tuple)    #will output whole tuple. (123,'hello')
print(tuple[0])    #will output first value. (123)
print(tuple + tuple1)    #will output (123,'hello','world')
tuple[1]='update'    #this will give you error.
```

Tipo de datos del diccionario

El diccionario consta de pares clave-valor. Está encerrado entre llaves {} y los valores se pueden asignar y acceder mediante corchetes [].

```
dic={'name':'red','age':10}
print(dic)    #will output all the key-value pairs. {'name':'red','age':10}
print(dic['name'])    #will output only value with 'name' key. 'red'
print(dic.values())    #will output list of values in dic. ['red',10]
print(dic.keys())    #will output list of keys. ['name','age']
```

Establecer tipos de datos

Los conjuntos son colecciones desordenadas de objetos únicos, hay dos tipos de conjuntos:

1. Conjuntos: son mutables y se pueden agregar nuevos elementos una vez que se definen los conjuntos

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)            # duplicates will be removed
> {'orange', 'banana', 'pear', 'apple'}
a = set('abracadabra')
print(a)                 # unique letters in a
> {'a', 'r', 'b', 'c', 'd'}
a.add('z')
print(a)
> {'a', 'c', 'r', 'b', 'z', 'd'}
```

2. Conjuntos congelados: son inmutables y no se pueden agregar nuevos elementos después de su definición.

```
b = frozenset('asdfagsa')
print(b)
> frozenset({'f', 'g', 'd', 'a', 's'})
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
print(cities)
> frozenset({'Frankfurt', 'Basel', 'Freiburg'})
```

Lea Tipos de datos de Python en línea: https://riptutorial.com/es/python/topic/9366/tipos-de-datos-de-python

Capítulo 192: Tipos de datos inmutables (int, float, str, tuple y frozensets)

Examples

Los caracteres individuales de las cadenas no son asignables

```
foo = "bar"
foo[0] = "c" # Error
```

El valor de la variable inmutable no se puede cambiar una vez que se crean.

Los miembros individuales de Tuple no son asignables

```
foo = ("bar", 1, "Hello!",)
foo[1] = 2 # ERROR!!
```

La segunda línea devolvería un error ya que los miembros de la tupla una vez creados no son asignables. Debido a la inmutabilidad de la tupla.

Los Frozenset son inmutables y no asignables.

```
foo = frozenset(["bar", 1, "Hello!"])
foo[2] = 7 # ERROR
foo.add(3) # ERROR
```

La segunda línea devolvería un error ya que los miembros de frozenset una vez creados no son asignables. La tercera línea devolvería el error, ya que los frozensets no admiten funciones que puedan manipular a los miembros.

Lea Tipos de datos inmutables (int, float, str, tuple y frozensets) en línea: https://riptutorial.com/es/python/topic/4806/tipos-de-datos-inmutables--int--float--str--tuple-y-frozensets-

Capítulo 193: tkinter

Introducción

Lanzado en Tkinter es la biblioteca GUI (interfaz gráfica de usuario) más popular de Python. Este tema explica el uso adecuado de esta biblioteca y sus características.

Observaciones

La capitalización del módulo tkinter es diferente entre Python 2 y 3. Para Python 2 use lo siguiente:

```
from Tkinter import *  # Capitalized
```

Para Python 3 usa lo siguiente:

```
from tkinter import *  # Lowercase
```

Para el código que funciona con Python 2 y 3, puede hacer

```
try:
    from Tkinter import *
except ImportError:
    from tkinter import *
```

o

```
from sys import version_info
if version_info.major == 2:
    from Tkinter import *
elif version_info.major == 3:
    from tkinter import *
```

> Ver la Documentación tkinter para más detalles.

Examples

Una aplicación tkinter mínima

`tkinter` es un kit de herramientas de GUI que proporciona un envoltorio alrededor de la biblioteca de GUI de Tk / Tcl y se incluye con Python. El siguiente código crea una nueva ventana usando `tkinter` y coloca algún texto en el cuerpo de la ventana.

> Nota: En Python 2, el uso de mayúsculas puede ser ligeramente diferente, consulte la sección de Comentarios a continuación.

```
import tkinter as tk

# GUI window is a subclass of the basic tkinter Frame object
class HelloWorldFrame(tk.Frame):
    def __init__(self, master):
        # Call superclass constructor
        tk.Frame.__init__(self, master)
        # Place frame into main window
        self.grid()
        # Create text box with "Hello World" text
        hello = tk.Label(self, text="Hello World! This label can hold strings!")
        # Place text box into frame
        hello.grid(row=0, column=0)

# Spawn window
if __name__ == "__main__":
    # Create main window object
    root = tk.Tk()
    # Set title of window
    root.title("Hello World!")
    # Instantiate HelloWorldFrame object
    hello_frame = HelloWorldFrame(root)
    # Start GUI
    hello_frame.mainloop()
```

Gerentes de geometría

Tkinter tiene tres mecanismos para la gestión de la geometría: `place` , `pack` y `grid` .

El `place` gestor utiliza coordenadas absolutas de píxeles.

El gestor de `pack` coloca los widgets en uno de los 4 lados. Los nuevos widgets se colocan al lado de los widgets existentes.

El administrador de `grid` coloca los widgets en una cuadrícula similar a una hoja de cálculo de cambio de tamaño dinámico.

Lugar

Los argumentos de palabras clave más comunes para `widget.place` son los siguientes:

- `x` , la coordenada x absoluta del widget
- `y` , la coordenada y absoluta del widget
- `height` , la altura absoluta del widget
- `width` , el ancho absoluto del widget

Un ejemplo de código usando `place` :

```
class PlaceExample(Frame):
    def __init__(self,master):
        Frame.__init__(self,master)
        self.grid()
        top_text=Label(master,text="This is on top at the origin")
```

```
        #top_text.pack()
        top_text.place(x=0,y=0,height=50,width=200)
        bottom_right_text=Label(master,text="This is at position 200,400")
        #top_text.pack()
        bottom_right_text.place(x=200,y=400,height=50,width=200)
# Spawn Window
if __name__=="__main__":
    root=Tk()
    place_frame=PlaceExample(root)
    place_frame.mainloop()
```

Paquete

`widget.pack` puede tomar los siguientes argumentos de palabras clave:

- `expand` , ya sea para llenar o no el espacio dejado por el padre
- `fill` , si desea expandir para llenar todo el espacio (NINGUNO (predeterminado), X, Y o AMBOS)
- `side` , el lado contra el que empacar (TOP (predeterminado), ABAJO, IZQUIERDO o DERECHO)

Cuadrícula

Los argumentos de palabras clave más utilizados de `widget.grid` son los siguientes:

- `row` , la fila del widget (por defecto el más pequeño desocupado)
- `rowspan` , el número de columnas que un widget abarca (valor predeterminado 1)
- `column` , la columna del widget (por defecto 0)
- `columnspan` , el número de columnas que abarca un widget (por defecto 1)
- `sticky` , dónde colocar el widget si la celda de la cuadrícula es más grande que él (combinación de N, NE, E, SE, S, SW, W, NW)

Las filas y columnas están indexadas a cero. Las filas aumentan bajando y las columnas aumentan yendo a la derecha.

Un ejemplo de código usando `grid` :

```
from tkinter import *

class GridExample(Frame):
    def __init__(self,master):
        Frame.__init__(self,master)
        self.grid()
        top_text=Label(self,text="This text appears on top left")
        top_text.grid() # Default position 0, 0
        bottom_text=Label(self,text="This text appears on bottom left")
        bottom_text.grid() # Default position 1, 0
        right_text=Label(self,text="This text appears on the right and spans both rows",
                         wraplength=100)
        # Position is 0,1
```

```
        # Rowspan means actual position is [0-1],1
        right_text.grid(row=0,column=1,rowspan=2)

# Spawn Window
if __name__=="__main__":
    root=Tk()
    grid_frame=GridExample(root)
    grid_frame.mainloop()
```

¡Nunca mezcle el `pack` y la `grid` dentro del mismo marco! ¡Al hacerlo se producirá un interbloqueo en la aplicación!

Lea tkinter en línea: https://riptutorial.com/es/python/topic/7574/tkinter

Capítulo 194: Trabajando alrededor del bloqueo global de intérpretes (GIL)

Observaciones

¿Por qué hay un GIL?

El GIL ha existido en CPython desde el inicio de los hilos de Python, en 1992. Está diseñado para garantizar la seguridad de los hilos al ejecutar el código de Python. Los intérpretes de Python escritos con un GIL impiden que varios subprocesos nativos ejecuten códigos de byte de Python a la vez. Esto facilita que los complementos se aseguren de que su código sea seguro para subprocesos: simplemente bloquee la GIL, y solo su subproceso activo puede ejecutarse, por lo que su código es automáticamente seguro para subprocesos.

Versión corta: la GIL garantiza que no importa cuántos procesadores y subprocesos tenga, *solo se ejecutará un subproceso de un intérprete de Python al mismo tiempo.*

Esto tiene muchos beneficios de facilidad de uso, pero también tiene muchos beneficios negativos.

Tenga en cuenta que un GIL no es un requisito del lenguaje Python. Por consiguiente, no puede acceder a la GIL directamente desde el código estándar de Python. No todas las implementaciones de Python utilizan un GIL.

Intérpretes que tienen un GIL: CPython, PyPy, Cython (pero puede desactivar el GIL con `nogil`)

Intérpretes que no tienen un GIL: Jython, IronPython

Detalles sobre cómo funciona el GIL:

Cuando se está ejecutando un hilo, bloquea la GIL. Cuando un hilo quiere ejecutarse, solicita el GIL y espera hasta que esté disponible. En CPython, antes de la versión 3.2, el hilo en ejecución verificaba después de un cierto número de instrucciones de Python para ver si otro código quería el bloqueo (es decir, liberó el bloqueo y luego lo solicitó nuevamente). Este método tendía a provocar la hambruna de hilos, en gran parte porque el hilo que liberaba el bloqueo lo volvería a adquirir antes de que los hilos en espera tuvieran la oportunidad de despertarse. Desde la versión 3.2, los subprocesos que desean que el GIL espere el bloqueo por algún tiempo, y después de ese tiempo, establecen una variable compartida que obliga al hilo en ejecución a ceder. Sin embargo, esto todavía puede resultar en tiempos de ejecución drásticamente más largos. Consulte los siguientes enlaces en dabeaz.com (en la sección de referencias) para obtener más detalles.

CPython libera automáticamente la GIL cuando un hilo realiza una operación de E / S. Las

bibliotecas de procesamiento de imágenes y las operaciones de procesamiento de números numpy liberan la GIL antes de realizar su procesamiento.

Beneficios de la GIL

Para los intérpretes que usan la GIL, la GIL es sistémica. Se utiliza para preservar el estado de la aplicación. Beneficios incluidos:

- Recolección de basura: los recuentos de referencia seguros para subprocesos deben modificarse mientras la GIL está bloqueada. *En CPython, toda la colección de garbarge está vinculada a la GIL.* Este es un grande; vea el artículo de la wiki de python.org sobre la GIL (que se encuentra en las Referencias, a continuación) para obtener detalles sobre lo que aún debe ser funcional si se desea eliminar la GIL.
- Facilidad para los programadores que tratan con GIL: bloquear todo es simplista, pero fácil de codificar
- Facilita la importación de módulos desde otros idiomas.

Consecuencias de la GIL

La GIL solo permite que un hilo ejecute el código de Python a la vez dentro del intérprete de python. Esto significa que el multihilo de procesos que ejecutan código de Python estricto simplemente no funciona. Al usar subprocesos contra GIL, es probable que tenga un peor rendimiento con los subprocesos que si se ejecutara en un solo subproceso.

Referencias:

https://wiki.python.org/moin/GlobalInterpreterLock - resumen rápido de lo que hace, detalles finos sobre todos los beneficios

http://programmers.stackexchange.com/questions/186889/why-was-python-written-with-the-gil - resumen claramente escrito

http://www.dabeaz.com/python/UnderstandingGIL.pdf : cómo funciona GIL y por qué se ralentiza en varios núcleos

http://www.dabeaz.com/GIL/gilvis/index.html : visualización de los datos que muestran cómo GIL bloquea los subprocesos

http://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/ - fácil de entender la historia del problema GIL

https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/ - detalles sobre las formas de solucionar las limitaciones de la GIL

Examples

Multiprocesamiento.Pool

La respuesta simple, cuando se pregunta cómo usar hilos en Python es: "No. Utilice procesos, en su lugar". El módulo de multiprocesamiento le permite crear procesos con una sintaxis similar a la creación de subprocesos, pero prefiero usar su conveniente objeto Pool.

Usando el código que David Beazley utilizó por primera vez para mostrar los peligros de los hilos en contra de la GIL , lo reescribiremos usando multiprocesamiento .

Código de David Beazley que mostraba problemas de subprocesos de GIL

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = Thread(target=countdown,args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Reescrita utilizando multiproceso.

```
import multiprocessing
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

start = time.time()
with multiprocessing.Pool as pool:
    pool.map(countdown, [COUNT/2, COUNT/2])

    pool.close()
    pool.join()

end = time.time()
print(end-start)
```

En lugar de crear hilos, esto crea nuevos procesos. Dado que cada proceso es su propio

intérprete, no hay colisiones de GIL. multiprocessing.Pool abrirá tantos procesos como núcleos haya en la máquina, aunque en el ejemplo anterior solo necesitaría dos. En un escenario del mundo real, desea diseñar su lista para que tenga al menos la misma longitud que procesadores en su máquina. El Pool ejecutará la función que le indica que ejecute con cada argumento, hasta el número de procesos que cree. Cuando la función finalice, cualquier función restante en la lista se ejecutará en ese proceso.

Descubrí que, incluso utilizando la instrucción `with` , si no cierra y se une al grupo, los procesos continúan existiendo. Para limpiar recursos, siempre cierro y me uno a mis piscinas.

Cython Nogil:

Cython es un intérprete alternativo de python. Utiliza el GIL, pero te permite deshabilitarlo. Ver su documentación

Como ejemplo, usando el código que David Beazley usó por primera vez para mostrar los peligros de los hilos contra la GIL , lo reescribiremos usando nogil:

Código de David Beazley que mostraba problemas de subprocesos de GIL

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = Thread(target=countdown,args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Reescrito usando nogil (SOLO FUNCIONA EN CYTHON):

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000
```

```
with nogil:
    t1 = Thread(target=countdown,args=(COUNT/2,))
    t2 = Thread(target=countdown,args=(COUNT/2,))
    start = time.time()
    t1.start();t2.start()
    t1.join();t2.join()

end = time.time()
print end-start
```

Es así de simple, siempre y cuando estés usando cython. Tenga en cuenta que la documentación indica que debe asegurarse de no cambiar ningún objeto de Python:

> El código en el cuerpo de la declaración no debe manipular los objetos de Python de ninguna manera, y no debe llamar a nada que manipule los objetos de Python sin volver a adquirir la GIL. Cython actualmente no comprueba esto.

Lea Trabajando alrededor del bloqueo global de intérpretes (GIL) en línea: https://riptutorial.com/es/python/topic/4061/trabajando-alrededor-del-bloqueo-global-de-interpretes--gil-

Capítulo 195: Trabajando con archivos ZIP

Sintaxis

- importar archivo zip
- *clase* zipfile. **ZipFile** (*archivo, modo = 'r', compresión = ZIP_STORED, allowZip64 = True*)

Observaciones

Si intenta abrir un archivo que no es un archivo ZIP, se `zipfile.BadZipFile` la excepción `zipfile.BadZipFile` .

En Python 2.7, este fue escrito `zipfile.BadZipfile` , y este nombre antiguo se conserva junto con el nuevo en Python 3.2+

Examples

Apertura de archivos zip

Para comenzar, importe el módulo `zipfile` y establezca el nombre del archivo.

```
import zipfile
filename = 'zipfile.zip'
```

Trabajar con archivos zip es muy similar a trabajar con archivos , usted crea el objeto abriendo el archivo zip, que le permite trabajar en él antes de cerrar el archivo nuevamente.

```
zip = zipfile.ZipFile(filename)
print(zip)
# <zipfile.ZipFile object at 0x0000000002E51A90>
zip.close()
```

En Python 2.7 y en Python 3 versiones superiores a 3.2, podemos usar el administrador de contexto `with` . Abrimos el archivo en modo "leer", y luego imprimimos una lista de nombres de archivos:

```
with zipfile.ZipFile(filename, 'r') as z:
    print(zip)
    # <zipfile.ZipFile object at 0x0000000002E51A90>
```

Examinando los contenidos de Zipfile

Hay algunas formas de inspeccionar el contenido de un archivo zip. Se puede utilizar el `printdir` que acaba de obtener una variedad de información que se envía a `stdout`

```
with zipfile.ZipFile(filename) as zip:
    zip.printdir()

    # Out:
    # File Name                                             Modified             Size
    # pyexpat.pyd                                    2016-06-25 22:13:34       157336
    # python.exe                                     2016-06-25 22:13:34        39576
    # python3.dll                                    2016-06-25 22:13:34        51864
    # python35.dll                                   2016-06-25 22:13:34      3127960
    # etc.
```

También podemos obtener una lista de nombres de archivo con la `namelist` método. Aquí, simplemente imprimimos la lista:

```
with zipfile.ZipFile(filename) as zip:
    print(zip.namelist())

# Out: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... etc. ...]
```

En lugar de `namelist` de `namelist` , podemos llamar al método `infolist` , que devuelve una lista de objetos `ZipInfo` , que contienen información adicional sobre cada archivo, por ejemplo, una marca de tiempo y el tamaño del archivo:

```
with zipfile.ZipFile(filename) as zip:
    info = zip.infolist()
    print(zip[0].filename)
    print(zip[0].date_time)
    print(info[0].file_size)

# Out: pyexpat.pyd
# Out: (2016, 6, 25, 22, 13, 34)
# Out: 157336
```

Extraer el contenido del archivo zip a un directorio.

Extraer todo el contenido de un archivo zip

```
import zipfile
with zipfile.ZipFile('zipfile.zip','r') as zfile:
    zfile.extractall('path')
```

Si desea extraer un solo archivo, use el método de extracción, toma la lista de nombres y la ruta como parámetro de entrada

```
import zipfile
f=open('zipfile.zip','rb')
zfile=zipfile.ZipFile(f)
for cont in zfile.namelist():
    zfile.extract(cont,path)
```

Creando nuevos archivos

Para crear un nuevo archivo, abra el archivo zip con el modo de escritura.

```
import zipfile
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

Para agregar archivos a este archivo use el método write ().

```
new_arch.write('filename.txt','filename_in_archive.txt') #first parameter is filename and
second parameter is filename in archive by default filename will taken if not provided
new_arch.close()
```

Si desea escribir una cadena de bytes en el archivo, puede usar el método writestr ().

```
str_bytes="string buffer"
new_arch.writestr('filename_string_in_archive.txt',str_bytes)
new_arch.close()
```

Lea Trabajando con archivos ZIP en línea: https://riptutorial.com/es/python/topic/3728/trabajando-con-archivos-zip

Capítulo 196: Trazado con matplotlib

Introducción

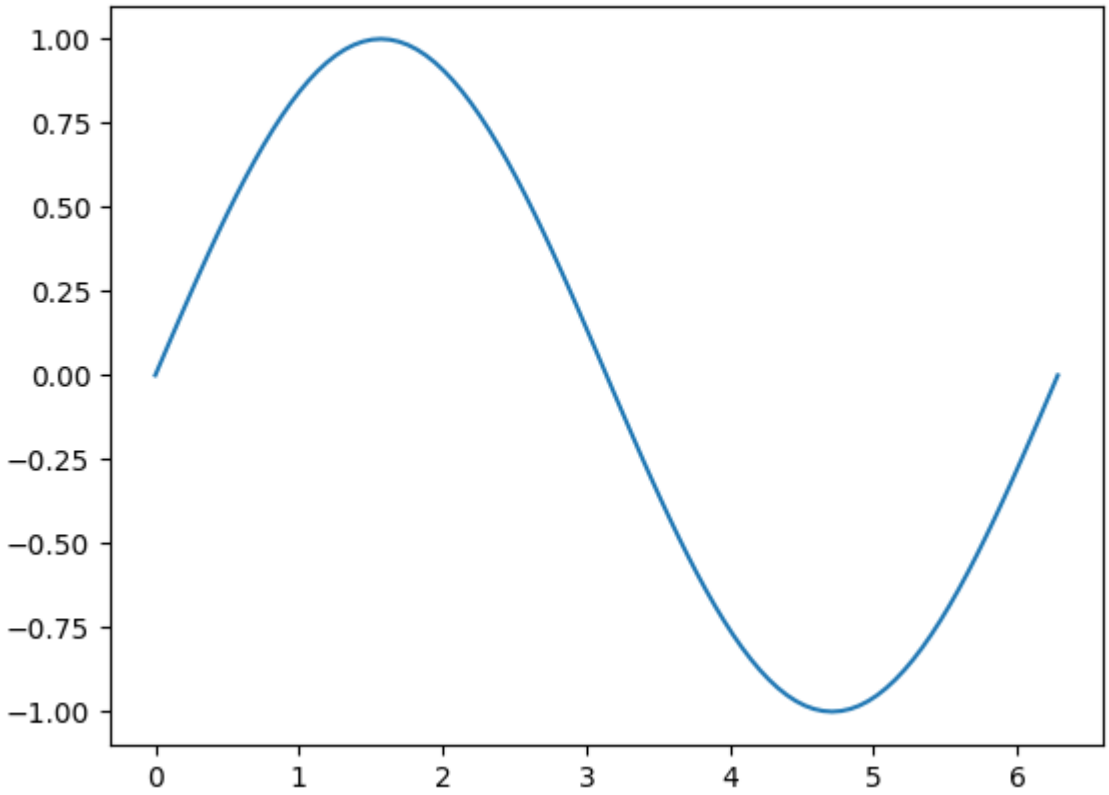
Matplotlib (https://matplotlib.org/) es una biblioteca para el trazado 2D basada en NumPy. Aquí hay algunos ejemplos básicos. Se pueden encontrar más ejemplos en la documentación oficial (https://matplotlib.org/2.0.2/gallery.html y https://matplotlib.org/2.0.2/examples/index.html) , así como en http: //www.riptutorial.com/topic/881

Examples

Una parcela simple en Matplotlib

Este ejemplo ilustra cómo crear una curva sinusoidal simple utilizando **Matplotlib**

```
# Plotting tutorials in Python
# Launching a simple plot

import numpy as np
import matplotlib.pyplot as plt

# angle varying between 0 and 2*pi
x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)                          # sine function

plt.plot(x, y)
plt.show()
```

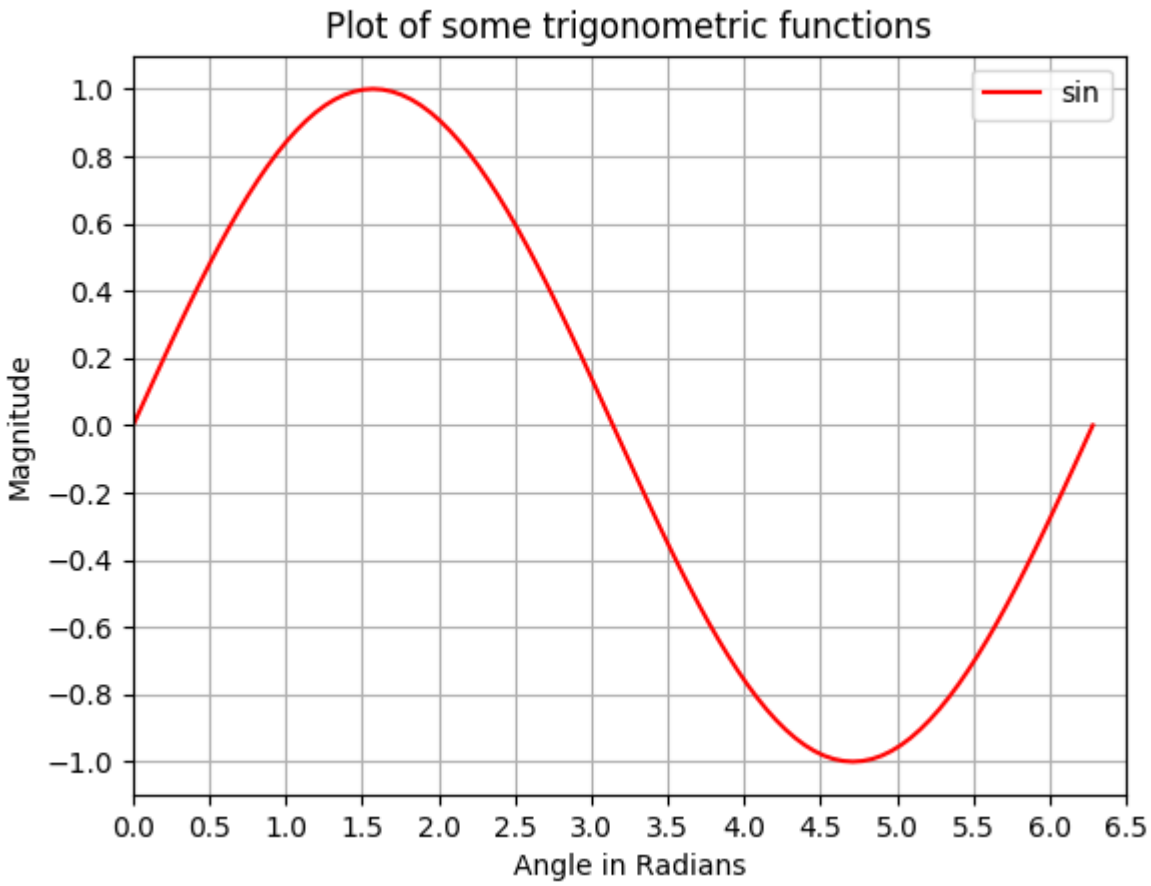


Agregar más características a un gráfico simple: etiquetas de eje, título, marcas de eje, cuadrícula y leyenda

En este ejemplo, tomamos una gráfica de curva sinusoidal y le agregamos más características; a saber, el título, etiquetas de eje, título, marcas de eje, cuadrícula y leyenda.

```
# Plotting tutorials in Python
# Enhancing a plot

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
```

```
plt.show()
```

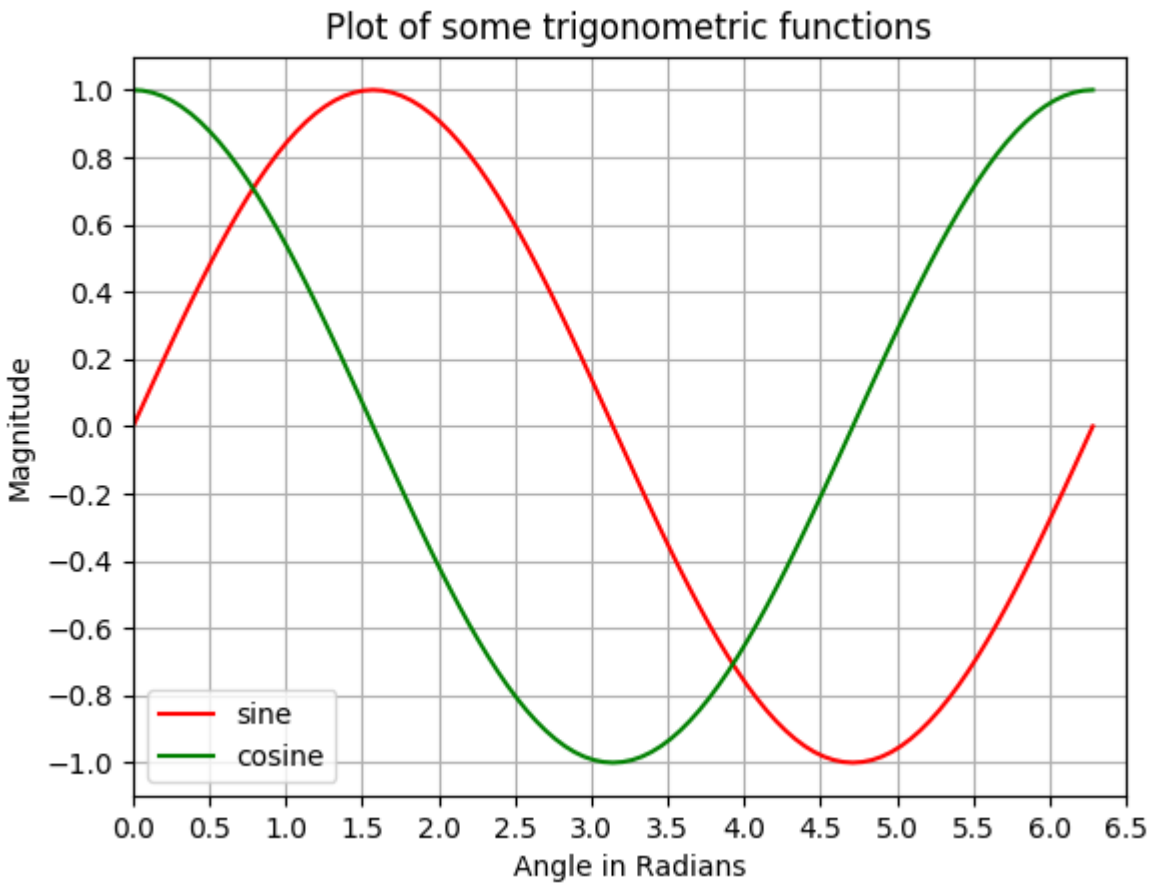


Haciendo múltiples parcelas en la misma figura por superposición similar a MATLAB

En este ejemplo, una curva sinusoidal y una curva de coseno se trazan en la misma figura mediante la superposición de los gráficos uno encima del otro.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using single plot command and legend

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, 'r', x, z, 'g') # r, g - red, green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
```

```
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend(['sine', 'cosine'])
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```

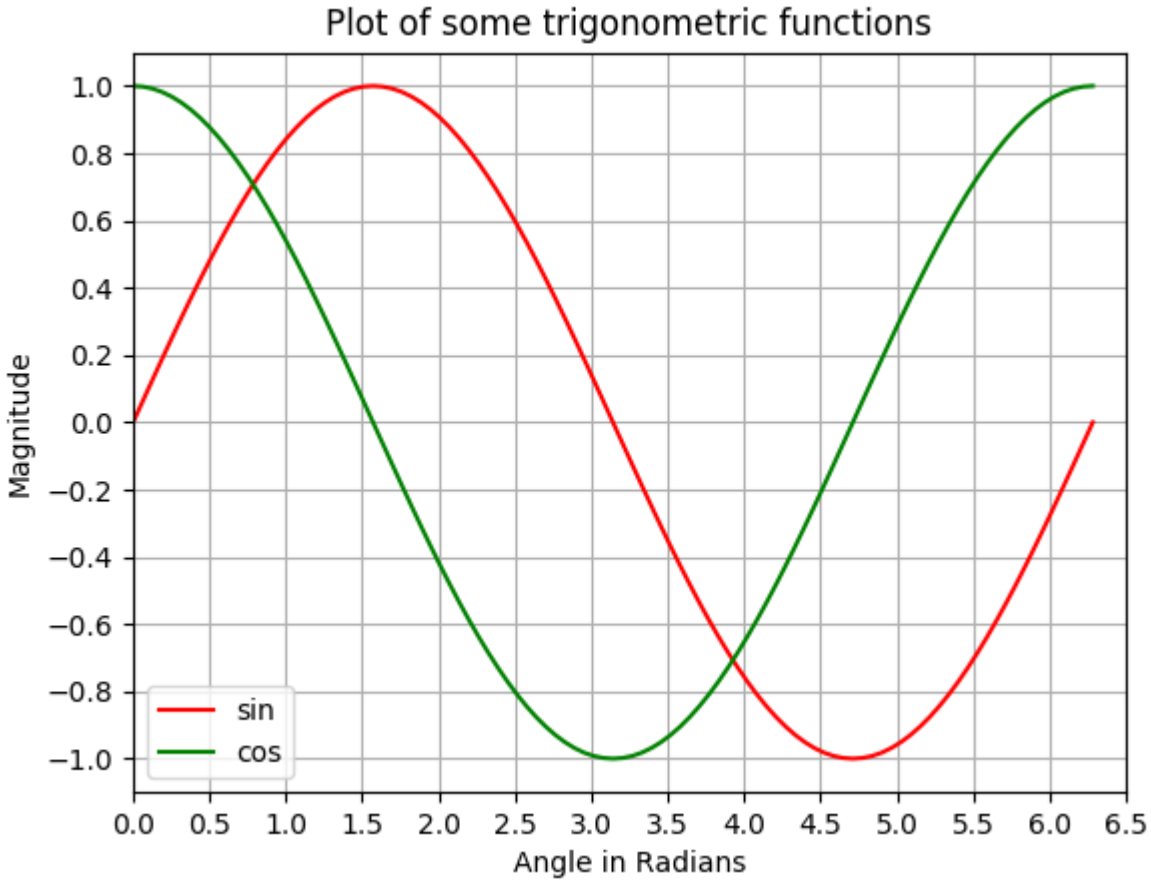


Realización de varios gráficos en la misma figura utilizando la superposición de gráficos con comandos de gráficos separados

Al igual que en el ejemplo anterior, aquí, una curva senoidal y una curva coseno se trazan en la misma figura utilizando comandos de trazado separados. Esto es más Pythonic y se puede usar para obtener identificadores separados para cada gráfico.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using multiple plot commands
# Much better and preferred than previous

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)
```

```
# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.plot(x, z, color='g', label='cos') # g - green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```

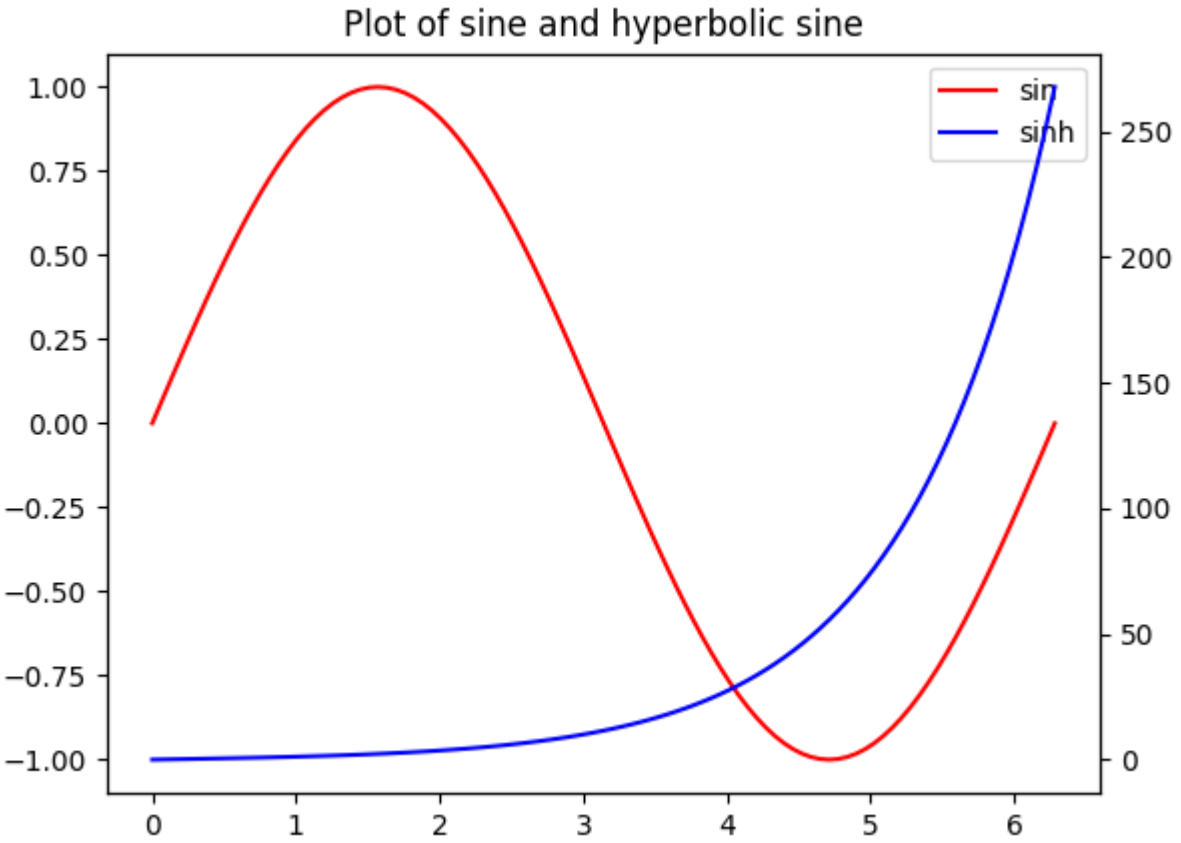


Gráficos con eje X común pero eje Y diferente: usando twinx ()

En este ejemplo, trazaremos una curva sinusoidal y una curva sinusoidal hiperbólica en la misma gráfica con un eje x común que tiene un eje y diferente. Esto se logra mediante el uso del **comando twinx ()** .

```
# Plotting tutorials in Python
# Adding Multiple plots by twin x axis
# Good for plots having different y axis range
# Separate axes and figure objects
```

```
# replicate axes object and plot curves
# use axes to set attributes

# Note:
# Grid for second curve unsuccessful : let me know if you find it! :(

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.sinh(x)

# separate the figure object and axes object
# from the plotting object
fig, ax1 = plt.subplots()

# Duplicate the axes with a different y axis
# and the same x axis
ax2 = ax1.twinx() # ax2 and ax1 will have common x axis and different y axis

# plot the curves on axes 1, and 2, and get the curve handles
curve1, = ax1.plot(x, y, label="sin", color='r')
curve2, = ax2.plot(x, z, label="sinh", color='b')

# Make a curves list to access the parameters in the curves
curves = [curve1, curve2]

# add legend via axes 1 or axes 2 object.
# one command is usually sufficient
# ax1.legend() # will not display the legend of ax2
# ax2.legend() # will not display the legend of ax1
ax1.legend(curves, [curve.get_label() for curve in curves])
# ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid

# Global figure properties
plt.title("Plot of sine and hyperbolic sine")
plt.show()
```

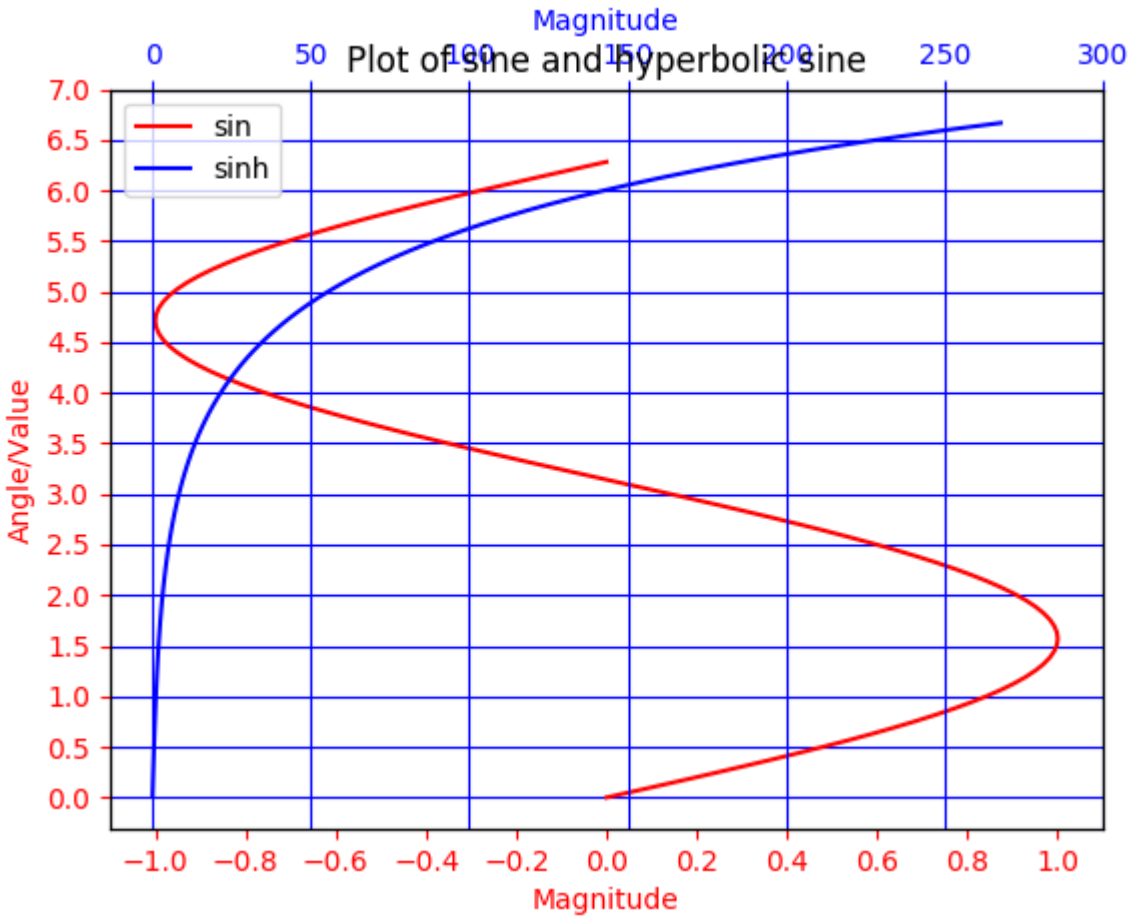


Gráficos con eje Y común y eje X diferente usando twiny ()

En este ejemplo, una gráfica con curvas que tienen un eje y común pero un eje x diferente se demuestra utilizando el método **twiny ()** . Además, algunas características adicionales como el título, la leyenda, las etiquetas, las cuadrículas, las marcas de eje y los colores se agregan a la trama.

```
# Plotting tutorials in Python
# Adding Multiple plots by twin y axis
# Good for plots having different x axis range
# Separate axes and figure objects
# replicate axes object and plot curves
# use axes to set attributes

import numpy as np
import matplotlib.pyplot as plt

y = np.linspace(0, 2.0*np.pi, 101)
x1 = np.sin(y)
x2 = np.sinh(y)

# values for making ticks in x and y axis
ynumbers = np.linspace(0, 7, 15)
xnumbers1 = np.linspace(-1, 1, 11)
xnumbers2 = np.linspace(0, 300, 7)

# separate the figure object and axes object
# from the plotting object
```

```
fig, ax1 = plt.subplots()

# Duplicate the axes with a different x axis
# and the same y axis
ax2 = ax1.twiny() # ax2 and ax1 will have common y axis and different x axis

# plot the curves on axes 1, and 2, and get the axes handles
curve1, = ax1.plot(x1, y, label="sin", color='r')
curve2, = ax2.plot(x2, y, label="sinh", color='b')

# Make a curves list to access the parameters in the curves
curves = [curve1, curve2]

# add legend via axes 1 or axes 2 object.
# one command is usually sufficient
# ax1.legend() # will not display the legend of ax2
# ax2.legend() # will not display the legend of ax1
# ax1.legend(curves, [curve.get_label() for curve in curves])
ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid

# x axis labels via the axes
ax1.set_xlabel("Magnitude", color=curve1.get_color())
ax2.set_xlabel("Magnitude", color=curve2.get_color())

# y axis label via the axes
ax1.set_ylabel("Angle/Value", color=curve1.get_color())
# ax2.set_ylabel("Magnitude", color=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# y ticks - make them coloured as well
ax1.tick_params(axis='y', colors=curve1.get_color())
# ax2.tick_params(axis='y', colors=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# x axis ticks via the axes
ax1.tick_params(axis='x', colors=curve1.get_color())
ax2.tick_params(axis='x', colors=curve2.get_color())

# set x ticks
ax1.set_xticks(xnumbers1)
ax2.set_xticks(xnumbers2)

# set y ticks
ax1.set_yticks(ynumbers)
# ax2.set_yticks(ynumbers) # also works

# Grids via axes 1 # use this if axes 1 is used to
# define the properties of common x axis
# ax1.grid(color=curve1.get_color())

# To make grids using axes 2
ax1.grid(color=curve2.get_color())
ax2.grid(color=curve2.get_color())
ax1.xaxis.grid(False)

# Global figure properties
plt.title("Plot of sine and hyperbolic sine")
plt.show()
```



Lea Trazado con matplotlib en línea: https://riptutorial.com/es/python/topic/10264/trazado-con-matplotlib

Capítulo 197: Tupla

Introducción

Una tupla es una lista inmutable de valores. Las tuplas son uno de los tipos de colección más simples y comunes de Python, y se pueden crear con el operador de coma (`value = 1, 2, 3`).

Sintaxis

- (1, a, "hola") # a debe ser una variable
- () # una tupla vacía
- (1,) # una tupla de 1 elemento. `(1)` no es una tupla.
- 1, 2, 3 # la tupla de 3 elementos (1, 2, 3)

Observaciones

Los paréntesis solo son necesarios para las tuplas vacías o cuando se usan en una llamada de función.

Una tupla es una secuencia de valores. Los valores pueden ser de cualquier tipo, y están indexados por números enteros, por lo que en este aspecto las tuplas se parecen mucho a las listas. La diferencia importante es que las tuplas son inmutables y hashable, por lo que se pueden usar en conjuntos y mapas

Examples

Tuplas de indexación

```
x = (1, 2, 3)
x[0]  # 1
x[1]  # 2
x[2]  # 3
x[3]  # IndexError: tuple index out of range
```

La indexación con números negativos comenzará desde el último elemento como -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Indexando una gama de elementos

```
print(x[:-1])   # (1, 2)
print(x[-1:])   # (3,)
print(x[1:3])   # (2, 3)
```

Las tuplas son inmutables

Una de las principales diferencias entre las `list` y las `tuple` en Python es que las tuplas son inmutables, es decir, no se pueden agregar o modificar elementos una vez que se inicializa la tupla. Por ejemplo:

```
>>> t = (1, 4, 9)
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

De manera similar, las tuplas no tienen los métodos `.append` y `.extend` como la `list` . Usar `+=` es posible, pero cambia el enlace de la variable, y no la tupla en sí:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Tenga cuidado al colocar objetos mutables, como `lists` , dentro de tuplas. Esto puede llevar a resultados muy confusos al cambiarlos. Por ejemplo:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Ayude ***a*** poner ***de*** un error y cambiar el contenido de la lista dentro de la tupla:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

Puede usar el operador `+=` para "agregar" a una tupla; esto funciona creando una nueva tupla con el nuevo elemento que "agregó" y asignándola a su variable actual; La tupla antigua no se cambia, pero se reemplaza!

Esto evita la conversión hacia y desde una lista, pero esto es lento y es una mala práctica, especialmente si se va a agregar varias veces.

Tuple son elementos sabios hashable y equiparables

```
hash( (1, 2) )  # ok
```

```
hash( ([], {"hello"})  # not ok, since lists and sets are not hashabe
```

Por lo tanto, una tupla se puede poner dentro de un `set` o como una clave en un `dict` solo si cada uno de sus elementos puede.

```
{ (1, 2) } #  ok
{ ([], {"hello"}) ) # not ok
```

Tupla

Sintácticamente, una tupla es una lista de valores separados por comas:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Aunque no es necesario, es común encerrar las tuplas entre paréntesis:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Crea una tupla vacía con paréntesis:

```
t0 = ()
type(t0)            # <type 'tuple'>
```

Para crear una tupla con un solo elemento, debe incluir una coma final:

```
t1 = 'a',
type(t1)              # <type 'tuple'>
```

Tenga en cuenta que un solo valor entre paréntesis no es una tupla:

```
t2 = ('a')
type(t2)              # <type 'str'>
```

Para crear una tupla de singleton es necesario tener una coma al final.

```
t2 = ('a',)
type(t2)              # <type 'tuple'>
```

Tenga en cuenta que para las tuplas singleton se recomienda (vea PEP8 en comas al final) usar paréntesis. Además, no hay espacios en blanco después de la coma final (ver PEP8 en espacios en blanco)

```
t2 = ('a',)           # PEP8-compliant
t2 = 'a',             # this notation is not recommended by PEP8
t2 = ('a', )          # this notation is not recommended by PEP8
```

Otra forma de crear una tupla es la función integrada `tuple` .

```
t = tuple('lupins')
print(t)              # ('l', 'u', 'p', 'i', 'n', 's')
t = tuple(range(3))
print(t)              # (0, 1, 2)
```

Estos ejemplos se basan en material del libro **Think Python** de Allen B. Downey .

Embalaje y desembalaje de tuplas

Las tuplas en Python son valores separados por comas. Los paréntesis que se incluyen para ingresar las tuplas son opcionales, por lo que las dos asignaciones

```
a = 1, 2, 3   # a is the tuple (1, 2, 3)
```

y

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

son equivalentes La asignación `a = 1, 2, 3` también se denomina *empaquetamiento* porque reúne valores en una tupla.

Tenga en cuenta que una tupla de un solo valor también es una tupla. Para decirle a Python que una variable es una tupla y no un solo valor, puedes usar una coma al final

```
a = 1  # a is the value 1
a = 1, # a is the tuple (1,)
```

También se necesita una coma si usa paréntesis

```
a = (1,) # a is the tuple (1,)
a = (1)  # a is the value 1 and not a tuple
```

Para desempaquetar valores de una tupla y hacer múltiples asignaciones use

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

El símbolo _ se puede usar como un nombre de variable desechable si solo se necesitan algunos elementos de una tupla, actuando como un marcador de posición:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Tuplas de un solo elemento:

```
x, = 1,  # x is the value 1
x  = 1,  # x is the tuple (1,)
```

En Python 3, una variable de destino con un prefijo `*` se puede usar como una variable de *captura* (ver Desempaquetando los iterables):

Python 3.x 3.0

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

Elementos de inversión

Invertir elementos dentro de una tupla

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

O usando reversa (invertida da un iterable que se convierte en una tupla):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Funciones de tupla incorporadas

Las tuplas soportan las siguientes funciones integradas

Comparación

Si los elementos son del mismo tipo, python realiza la comparación y devuelve el resultado. Si los elementos son de diferentes tipos, verifica si son números.

- Si son números, realice la comparación.
- Si cualquiera de los elementos es un número, se devuelve el otro elemento.
- De lo contrario, los tipos se ordenan alfabéticamente.

Si llegamos al final de una de las listas, la lista más larga es "más grande". Si ambas listas son iguales devuelve 0.

```
tuple1 = ('a', 'b', 'c', 'd', 'e')
tuple2 = ('1','2','3')
tuple3 = ('a', 'b', 'c', 'd', 'e')
```

```
cmp(tuple1, tuple2)
Out: 1

cmp(tuple2, tuple1)
Out: -1

cmp(tuple1, tuple3)
Out: 0
```

Longitud de la tupla

La función `len` devuelve la longitud total de la tupla.

```
len(tuple1)
Out: 5
```

Max de una tupla

La función `max` devuelve el elemento de la tupla con el valor máximo

```
max(tuple1)
Out: 'e'

max(tuple2)
Out: '3'
```

Min de una tupla

La función min devuelve el elemento de la tupla con el valor mínimo

```
min(tuple1)
Out: 'a'

min(tuple2)
Out: '1'
```

Convertir una lista en tupla

La función integrada `tuple` convierte una lista en una tupla.

```
list = [1,2,3,4,5]
tuple(list)
Out: (1, 2, 3, 4, 5)
```

Concatenación de tuplas

Usa `+` para concatenar dos tuplas

```
tuple1 + tuple2
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

Lea Tupla en línea: https://riptutorial.com/es/python/topic/927/tupla

Capítulo 198: Unicode

Examples

Codificación y decodificación.

Codifique siempre de unicode a bytes. En esta dirección, **puedes elegir la codificación** .

```
>>> u'🐍'.encode('utf-8')
'\xf0\x9f\x90\x8d'
```

La otra forma es *decodificar* de bytes a unicode. En esta dirección, **tienes que saber qué es la codificación** .

```
>>> b'\xf0\x9f\x90\x8d'.decode('utf-8')
u'\U0001f40d'
```

Lea Unicode en línea: https://riptutorial.com/es/python/topic/5618/unicode

Capítulo 199: Unicode y bytes

Sintaxis

- str.encode (codificación, errores = 'estricto')
- bytes.decode (codificación, errores = 'estricto')
- abierto (nombre de archivo, modo, codificación = Ninguno)

Parámetros

Parámetro	Detalles
codificación	La codificación a utilizar, por ejemplo, `'ascii'` , `'utf8'` , etc ...
errores	El modo de errores, por ejemplo, `'replace'` para reemplazar los caracteres incorrectos con signos de interrogación, `'ignore'` para ignorar los caracteres incorrectos, etc.

Examples

Lo esencial

En Python 3, `str` es el tipo para cadenas habilitadas para Unicode, mientras que `bytes` es el tipo para secuencias de bytes sin procesar.

```
type("f") == type(u"f")  # True, <class 'str'>
type(b"f")               # <class 'bytes'>
```

En Python 2, una cadena casual era una secuencia de bytes sin procesar por defecto y la cadena Unicode era cada cadena con el prefijo "u".

```
type("f") == type(b"f")  # True, <type 'str'>
type(u"f")               # <type 'unicode'>
```

Unicode a bytes

Las cadenas Unicode se pueden convertir a bytes con `.encode(encoding)` .

Python 3

```
>>> "£13.55".encode('utf8')
```

```
b'\xc2\xa313.55'
>>> "£13.55".encode('utf16')
b'\xff\xfe\xa3\x001\x003\x00.\x005\x005\x00'
```

Python 2

en py2, la codificación de la consola predeterminada es `sys.getdefaultencoding() == 'ascii'` y no `utf-8` como en py3, por lo tanto, imprimirla como en el ejemplo anterior no es directamente posible.

```
>>> print type(u"£13.55".encode('utf8'))
<type 'str'>
>>> print u"£13.55".encode('utf8')
SyntaxError: Non-ASCII character '\xc2' in...

# with encoding set inside a file

# -*- coding: utf-8 -*-
>>> print u"£13.55".encode('utf8')
┬ú13.55
```

Si la codificación no puede manejar la cadena, se genera un `UnicodeEncodeError`:

```
>>> "£13.55".encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xa3' in position 0: ordinal not in
range(128)
```

Bytes a Unicode

Los bytes se pueden convertir en cadenas Unicode con `.decode(encoding)` .

¡Una secuencia de bytes solo se puede convertir en una cadena Unicode a través de la codificación apropiada!

```
>>> b'\xc2\xa313.55'.decode('utf8')
'£13.55'
```

Si la codificación no puede manejar la cadena, se `UnicodeDecodeError` un `UnicodeDecodeError` :

```
>>> b'\xc2\xa313.55'.decode('utf16')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/csaftoiu/csaftoiu-github/yahoo-groups-
backup/.virtualenv/bin/../lib/python3.5/encodings/utf_16.py", line 16, in decode
    return codecs.utf_16_decode(input, errors, True)
UnicodeDecodeError: 'utf-16-le' codec can't decode byte 0x35 in position 6: truncated data
```

Codificación / decodificación de manejo de errores.

`.encode` y `.decode` tienen modos de error.

El valor predeterminado es `'strict'` , que genera excepciones en caso de error. Otros modos son más indulgentes.

Codificación

```
>>> "£13.55".encode('ascii', errors='replace')
b'?13.55'
>>> "£13.55".encode('ascii', errors='ignore')
b'13.55'
>>> "£13.55".encode('ascii', errors='namereplace')
b'\\N{POUND SIGN}13.55'
>>> "£13.55".encode('ascii', errors='xmlcharrefreplace')
b'&#163;13.55'
>>> "£13.55".encode('ascii', errors='backslashreplace')
b'\\xa313.55'
```

Descodificación

```
>>> b = "£13.55".encode('utf8')
>>> b.decode('ascii', errors='replace')
'��13.55'
>>> b.decode('ascii', errors='ignore')
'13.55'
>>> b.decode('ascii', errors='backslashreplace')
'\\xc2\\xa313.55'
```

Moral

De lo anterior se desprende claramente que es vital mantener sus codificaciones rectas cuando se trata de unicode y bytes.

Archivo I / O

Los archivos abiertos en un modo no binario (por ejemplo, `'r'` o `'w'`) tratan con cadenas. La codificación sordera es `'utf8'` .

```
open(fn, mode='r')                    # opens file for reading in utf8
open(fn, mode='r', encoding='utf16')  # opens file for reading utf16

# ERROR: cannot write bytes when a string is expected:
open("foo.txt", "w").write(b"foo")
```

Los archivos abiertos en modo binario (por ejemplo, `'rb'` o `'wb'`) tratan con bytes. No se puede

especificar ningún argumento de codificación ya que no hay codificación.

```
open(fn, mode='wb')  # open file for writing bytes

# ERROR: cannot write string when bytes is expected:
open(fn, mode='wb').write("hi")
```

Lea Unicode y bytes en línea: https://riptutorial.com/es/python/topic/1216/unicode-y-bytes

Capítulo 200: urllib

Examples

HTTP GET

Python 2.x 2.7

Python 2

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

El uso de `urllib.urlopen()` devolverá un objeto de respuesta, que se puede manejar de manera similar a un archivo.

```
print response.code
# Prints: 200
```

El `response.code` representa el valor de retorno de http. 200 está bien, 404 es NotFound, etc.

```
print response.read()
'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack. etc'
```

`response.read()` y `response.readlines()` se pueden usar para leer el archivo html real devuelto por la solicitud. Estos métodos funcionan de manera similar a `file.read*`

Python 3.x 3.0

Python 3

```
import urllib.request

print(urllib.request.urlopen("http://stackoverflow.com/documentation/"))
# Prints: <http.client.HTTPResponse at 0x7f37a97e3b00>

response = urllib.request.urlopen("http://stackoverflow.com/documentation/")

print(response.code)
# Prints: 200
print(response.read())
# Prints: b'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack
Overflow</title>
```

El módulo se ha actualizado para Python 3.x, pero los casos de uso siguen siendo básicamente los mismos. `urllib.request.urlopen` devolverá un objeto similar a un archivo similar.

POST HTTP

Para POST, los datos pasan los argumentos de consulta codificados como datos a urlopen ()

Python 2.x 2.7

Python 2

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.urlencode(query_parms)
response = urllib.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: '<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Log In - Stack Overflow'
```

Python 3.x 3.0

Python 3

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.parse.urlencode(query_parms).encode('utf-8')
response = urllib.request.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: b'<!DOCTYPE html>\r\n<html>....etc'
```

Decodificar bytes recibidos de acuerdo a la codificación del tipo de contenido

Los bytes recibidos deben decodificarse con la codificación de caracteres correcta para interpretarlos como texto:

Python 3.x 3.0

```
import urllib.request

response = urllib.request.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Python 2.x 2.7

```
import urllib2
response = urllib2.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().getencoding()
```

```
html = data.decode(encoding)
```

Lea urllib en línea: https://riptutorial.com/es/python/topic/2645/urllib

Capítulo 201: Usando bucles dentro de funciones

Introducción

En Python, la función se devolverá tan pronto como la ejecución llegue a la declaración de "retorno".

Examples

Declaración de retorno dentro del bucle en una función

En este ejemplo, la función regresará tan pronto como el valor var tenga 1

```
def func(params):
    for value in params:
        print ('Got value {}'.format(value))

        if value == 1:
            # Returns from function as soon as value is 1
            print (">>>> Got 1")
            return

        print ("Still looping")

    return "Couldn't find 1"

func([5, 3, 1, 2, 8, 9])
```

salida

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>>> Got 1
```

Lea Usando bucles dentro de funciones en línea:
https://riptutorial.com/es/python/topic/10883/usando-bucles-dentro-de-funciones

Capítulo 202: Uso del módulo "pip": PyPI Package Manager

Introducción

A veces es posible que necesite usar el administrador de paquetes pip dentro de Python, por ejemplo. cuando algunas importaciones pueden aumentar `ImportError` y desea manejar la excepción. Si desembala en Windows `Python_root/Scripts/pip.exe` dentro del archivo `__main__.py` se almacena, donde se importa la clase `main` del paquete `pip` . Esto significa que el paquete pip se usa siempre que uses el ejecutable pip. Para el uso de pip como ejecutable, consulte: pip: PyPI Package Manager

Sintaxis

- pip. <function | attribute | class> donde function es uno de:
  - autocompletar ()
    - Finalización de comandos y opciones para el analizador de opciones principal (y opciones) y sus subcomandos (y opciones). Habilite mediante la obtención de uno de los scripts de shell de finalización (bash, zsh o fish).
  - check_isolated (args)
    - param args {lista}
    - devuelve {booleano}
  - create_main_parser ()
    - devuelve el objeto {pip.baseparser.ConfigOptionParser}
  - main (args = None)
    - param args {lista}
    - devuelve {entero} Si no falla, devuelve 0
  - parseopts (args)
    - param args {lista}
  - get_installed_distributions ()
    - devuelve {lista}
  - get_similar_commands (nombre)
    - Nombre del comando auto-correcto.
    - nombre de parámetro {cadena}
    - devuelve {booleano}
  - get_summaries (ordenado = Verdadero)
    - Rendimientos ordenados (nombre del comando, resumen del comando) tuplas.
  - get_prog ()
    - devuelve {cadena}
  - dist_is_editable (dist)
    - ¿Es la distribución una instalación editable?
    - param dist {objeto}
    - devuelve {booleano}

- comandos_dicto
  - atributo {diccionario}

Examples

Ejemplo de uso de comandos

```
import pip

command = 'install'
parameter = 'selenium'
second_param = 'numpy' # You can give as many package names as needed
switch = '--upgrade'

pip.main([command, parameter, second_param, switch])
```

Solo los parámetros necesarios son obligatorios, por lo que tanto `pip.main(['freeze'])` como `pip.main(['freeze', '', ''])` son aceptables.

Instalación por lotes

Es posible pasar muchos nombres de paquetes en una llamada, pero si falla una instalación / actualización, todo el proceso de instalación se detiene y termina con el estado '1'.

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

Si no desea detenerse cuando fallan algunas instalaciones, llame a la instalación en bucle.

```
for i in installed:
        pip.main(['install']+i.key+['--upgrade'])
```

Manejo de la excepción de ImportError

Cuando usa el archivo python como módulo, no es necesario verificar siempre si el paquete está instalado, pero sigue siendo útil para los scripts.

```
if __name__ == '__main__':
    try:
        import requests
    except ImportError:
        print("To use this module you need 'requests' module")
        t = input('Install requests? y/n: ')
        if t == 'y':
            import pip
            pip.main(['install', 'requests'])
```

```
            import requests
            import os
            import sys
            pass
        else:
            import os
            import sys
            print('Some functionality can be unavailable.')
else:
    import requests
    import os
    import sys
```

Fuerza de instalación

Muchos paquetes, por ejemplo, en la versión 3.4 se ejecutarán en 3.6 muy bien, pero si no hay distribuciones para una plataforma específica, no se pueden instalar, pero hay una solución alternativa. En la convención de nombres de los archivos .whl (conocidos como ruedas), decida si puede instalar el paquete en la plataforma especificada. P.ej. `scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl` [nombre_paquete] - [versión] - [intérprete de python] - [intérprete de python] - [Sistema operativo] .whl. Si se cambia el nombre del archivo de rueda, para que la plataforma coincida, pip intenta instalar el paquete incluso si la plataforma o la versión de python no coinciden. Eliminar la plataforma o el intérprete del nombre generará un error en la versión más reciente del módulo pip `kjhfkjdf.whl is not a valid wheel filename.` .

Alternativamente, el archivo .whl se puede desempaquetar usando un archivador como 7-zip. - Por lo general, contiene la meta carpeta de distribución y la carpeta con archivos de origen. Estos archivos de origen se pueden desempaquetar simplemente en el directorio de `site-packges` menos que esta rueda contenga el script de instalación, de ser así, debe ejecutarse primero.

Lea Uso del módulo "pip": PyPI Package Manager en línea: https://riptutorial.com/es/python/topic/10730/uso-del-modulo--pip---pypi-package-manager

Capítulo 203: Velocidad de Python del programa.

Examples

Notación

Idea básica

La notación utilizada al describir la velocidad de su programa Python se llama notación Big-O. Digamos que tienes una función:

```
def list_check(to_check, the_list):
    for item in the_list:
        if to_check == item:
          return True
    return False
```

Esta es una función simple para verificar si un elemento está en una lista. Para describir la complejidad de esta función, dirá O (n). Esto significa "Orden de n", ya que la función O se conoce como la función Orden.

O (n): generalmente n es el número de elementos en el contenedor

O (k): generalmente k es el valor del parámetro o el número de elementos en el parámetro

Lista de operaciones

Operaciones: Caso promedio (se supone que los parámetros se generan aleatoriamente)

Anexo: O (1)

Copia: O (n)

Del slice: O (n)

Eliminar elemento: O (n)

Insertar: O (n)

Obtener artículo: O (1)

Elemento de ajuste: O (1)

Iteración: O (n)

Obtener rebanada: O (k)

Establecer rebanada: O (n + k)

Extender: O (k)

Ordenar: O (n log n)

Multiplicar: O (nk)

x en s: O (n)

min (s), max (s): O (n)

Obtener longitud: O (1)

Operaciones de deque

Un deque es una cola de doble final.

```
class Deque:
def __init__(self):
    self.items = []

def isEmpty(self):
    return self.items == []

def addFront(self, item):
    self.items.append(item)

def addRear(self, item):
    self.items.insert(0,item)

def removeFront(self):
    return self.items.pop()

def removeRear(self):
    return self.items.pop(0)

def size(self):
    return len(self.items)
```

Operaciones: Caso promedio (se supone que los parámetros se generan aleatoriamente)

Anexo: O (1)

Apéndice: O (1)

Copia: O (n)

Extender: O (k)

Extendente: O (k)

Pop: O (1)

Personas: O (1)

Eliminar: O (n)

Girar: O (k)

Establecer operaciones

Operación: Caso promedio (asume parámetros generados aleatoriamente): Peor caso

x en s: O (1)

Diferencia s - t: O (len (s))

Intersección s & t: O (min (len (s), len (t))): O (len (s) * len (t)

Intersección múltiple s1 & s2 & s3 & ... & sn:: (n-1) * O (l) donde l es max (len (s1), ..., len (sn))

s.difference_update (t): O (len (t)): O (len (t) * len (s))

s.symetric_difference_update (t): O (len (t))

Diferencia simétrica s ^ t: O (len (s)): O (len (s) * len (t))

Unión s | t: O (len (s) + len (t))

Notaciones algorítmicas ...

Hay ciertos principios que se aplican a la optimización en cualquier lenguaje de computadora, y Python no es una excepción. **No optimice a medida que avanza** : escriba su programa sin importar las posibles optimizaciones, concentrándose en asegurarse de que el código sea limpio, correcto y comprensible. Si es demasiado grande o demasiado lento cuando haya terminado, entonces puede considerar optimizarlo.

Recuerde la regla 80/20 : en muchos campos puede obtener el 80% del resultado con el 20% del esfuerzo (también llamada regla 90/10 - depende de con quién hable). Cuando esté a punto de optimizar el código, use la creación de perfiles para averiguar a dónde va ese 80% del tiempo de ejecución, para que sepa dónde concentrar su esfuerzo.

Ejecute siempre los puntos de referencia "antes" y "después" : ¿De qué otra manera sabrá que sus optimizaciones realmente hicieron una diferencia? Si su código optimizado resulta ser solo un poco más rápido o más pequeño que la versión original, deshaga los cambios y vuelva al código original y sin cifrar.

Use los algoritmos y las estructuras de datos correctos: No use un algoritmo de clasificación de burbuja O (n2) para ordenar mil elementos cuando haya una ordenación rápida O (n log n) disponible. Del mismo modo, no almacene mil elementos en una matriz que requiera una búsqueda O (n) cuando podría usar un árbol binario O (log n) o una tabla hash O (1) de Python.

Para más información, visite el siguiente enlace ... Python Speed Up

Las siguientes 3 notaciones asintóticas se usan principalmente para representar la complejidad

del tiempo de los algoritmos.

1. **Θ Notación** : la notación theta limita las funciones desde arriba y desde abajo, por lo que define el comportamiento asintótico exacto. Una forma sencilla de obtener la notación Theta de una expresión es eliminar los términos de orden inferior e ignorar las constantes iniciales. Por ejemplo, considere la siguiente expresión. 3n3 + 6n2 + 6000 = Θ (n3) La eliminación de los términos de orden inferior siempre está bien porque siempre habrá un n0, luego de lo cual Θ (n3) tiene valores más altos que Θn2) independientemente de las constantes involucradas. Para una función dada g (n), denotamos que Θ (g (n)) está siguiendo un conjunto de funciones. Θ (g (n)) = {f (n): existen constantes positivas c1, c2 y n0 tales que 0 <= c1 *g (n)* <= *f (n)* <= *c2* g (n) para todo n> = n0} La definición anterior significa que si f (n) es theta de g (n), entonces el valor f (n) siempre está entre c1 *g (n) y c2* g (n) para valores grandes de n (n> = n0). La definición de theta también requiere que f (n) no sea negativa para valores de n mayores que n0.

2. **Notación Big O** : La notación Big O define un límite superior de un algoritmo, limita una función solo desde arriba. Por ejemplo, considérese el caso de Insertion Sort. Lleva tiempo lineal en el mejor de los casos y el tiempo cuadrático en el peor de los casos. Podemos decir con seguridad que la complejidad temporal de la ordenación de inserción es O (n ^ 2). Tenga en cuenta que O (n ^ 2) también cubre el tiempo lineal. Si usamos la notación para representar la complejidad de tiempo de la ordenación por Inserción, tenemos que usar dos afirmaciones para el mejor y el peor de los casos:

1. La complejidad en el peor de los casos de Insertion Sort es Θ (n ^ 2).
2. El mejor caso de complejidad en el tiempo de Insertion Sort es Θ (n).

La notación Big O es útil cuando solo tenemos un límite superior en la complejidad de tiempo de un algoritmo. Muchas veces encontramos fácilmente un límite superior simplemente mirando el algoritmo. O (g (n)) = {f (n): existen constantes positivas c y n0 tales que 0 <= f (n) <= cg (n) para todos n> = n0}

3. **Ω Notación** : al igual que la notación Big O proporciona un límite superior asintótico en una función, la notación provides proporciona un límite inferior asintótico. Ω La notación <puede ser útil cuando tenemos un límite inferior en la complejidad del tiempo de un algoritmo. Como se discutió en la publicación anterior, el mejor rendimiento de un algoritmo generalmente no es útil, la notación Omega es la notación menos utilizada entre las tres. Para una función dada g (n), denotamos por Ω (g (n)) el conjunto de funciones. Ω (g (n)) = {f (n): existen constantes positivas c y n0 tales que 0 <= cg (n) <= f (n) para todos n> = n0}. Consideremos el mismo ejemplo de orden de inserción aquí. La complejidad temporal de la clasificación de inserción se puede escribir como Ω (n), pero no es una información muy útil sobre la clasificación de inserción, ya que generalmente estamos interesados en el peor de los casos y, a veces, en el caso promedio.

Lea Velocidad de Python del programa. en línea:
https://riptutorial.com/es/python/topic/9185/velocidad-de-python-del-programa-

Capítulo 204: Visualización de datos con Python

Examples

Matplotlib

Matplotlib es una biblioteca de trazado matemático para Python que proporciona una variedad de funciones de trazado diferentes.

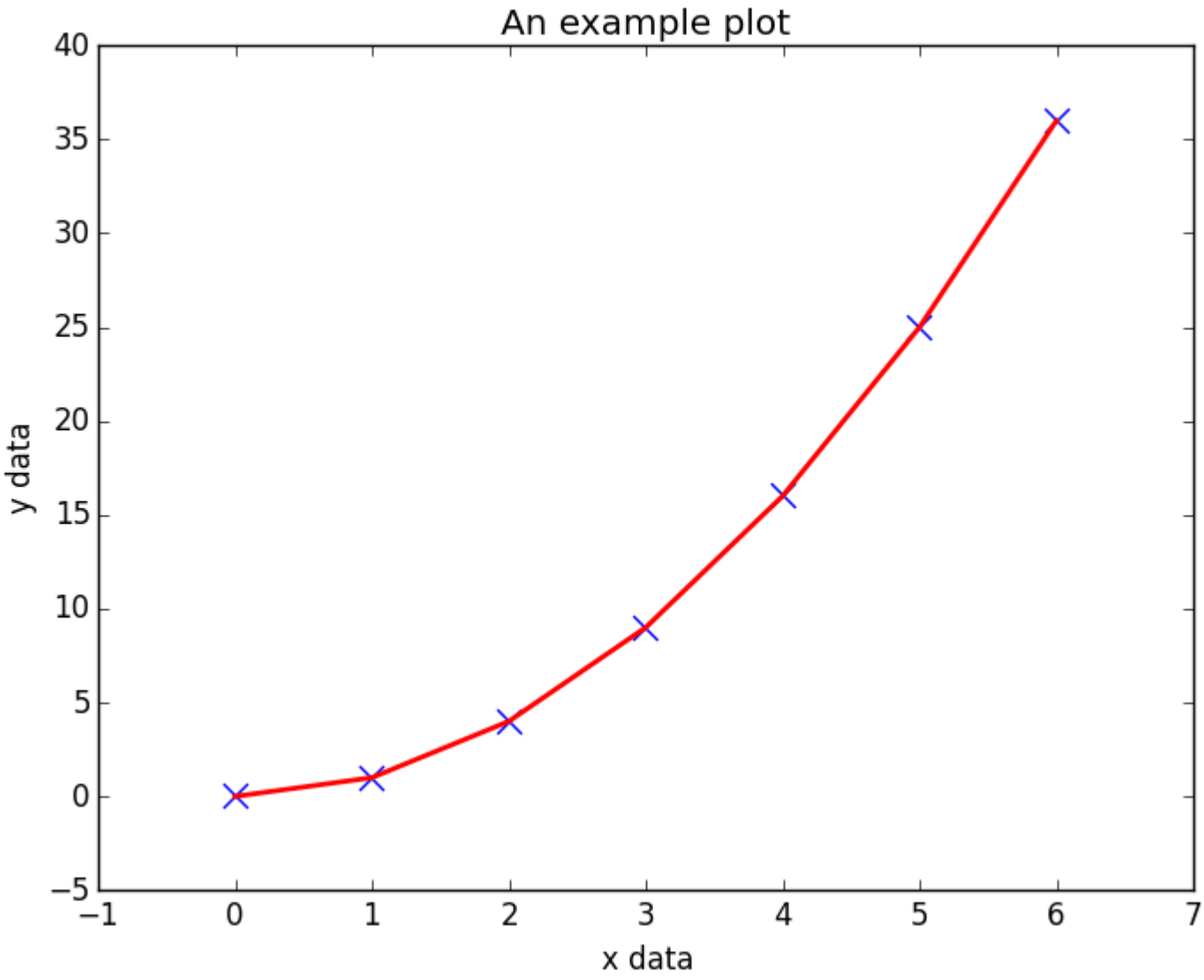
La documentación de matplotlib se puede encontrar aquí , con los SO Docs disponibles aquí .

Matplotlib proporciona dos métodos distintos para trazar, aunque son intercambiables en su mayor parte:

- En primer lugar, matplotlib proporciona la interfaz `pyplot` , una interfaz directa y fácil de usar que permite trazar gráficos complejos en un estilo similar a MATLAB.
- En segundo lugar, matplotlib permite al usuario controlar los diferentes aspectos (ejes, líneas, tics, etc.) directamente mediante un sistema basado en objetos. Esto es más difícil pero permite un control completo sobre toda la trama.

A continuación se muestra un ejemplo del uso de la interfaz `pyplot` para trazar algunos datos generados:

```
import matplotlib.pyplot as plt

# Generate some data for plotting.
x = [0, 1, 2, 3, 4, 5, 6]
y = [i**2 for i in x]

# Plot the data x, y with some keyword arguments that control the plot style.
# Use two different plot commands to plot both points (scatter) and a line (plot).

plt.scatter(x, y, c='blue', marker='x', s=100) # Create blue markers of shape "x" and size 100
plt.plot(x, y, color='red', linewidth=2) # Create a red line with linewidth 2.

# Add some text to the axes and a title.
plt.xlabel('x data')
plt.ylabel('y data')
plt.title('An example plot')

# Generate the plot and show to the user.
plt.show()
```



Tenga en cuenta que se `plt.show()` que `plt.show()` es problemático en algunos entornos debido a la ejecución de `matplotlib.pyplot` en modo interactivo, y si es así, el comportamiento de bloqueo se puede anular explícitamente al pasar un argumento opcional, `plt.show(block=True)` , para paliar el problema.

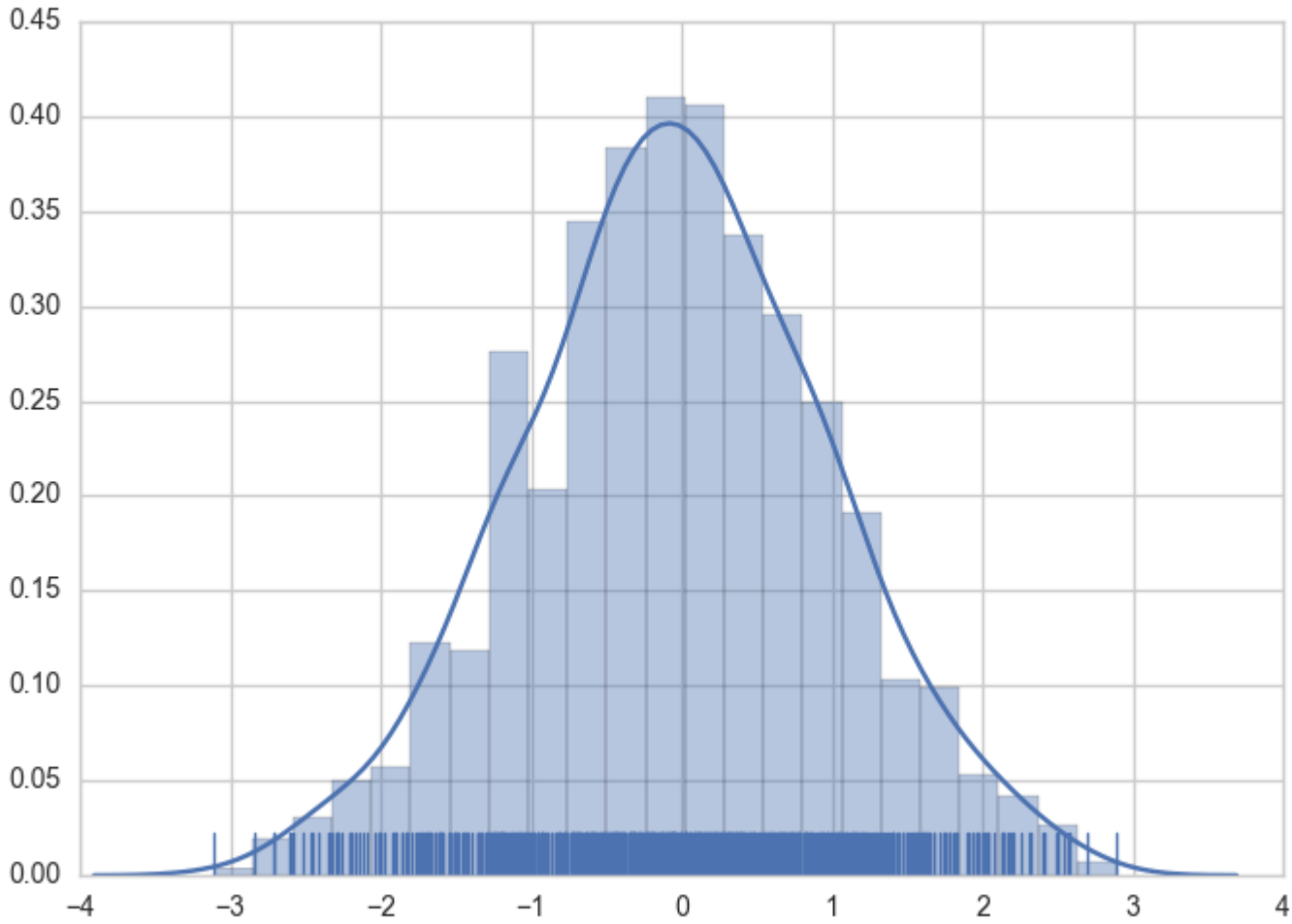
Seaborn

Seaborn es una envoltura alrededor de Matplotlib que facilita la creación de gráficos estadísticos comunes. La lista de gráficos admitidos incluye gráficos de distribución univariada y bivariada, gráficos de regresión y varios métodos para representar variables categóricas. La lista completa de las parcelas que proporciona Seaborn se encuentra en su referencia API .

Crear gráficos en Seaborn es tan simple como llamar a la función de gráficos apropiada. Este es un ejemplo de la creación de un histograma, una estimación de la densidad del kernel y una gráfica de tap para datos generados aleatoriamente.

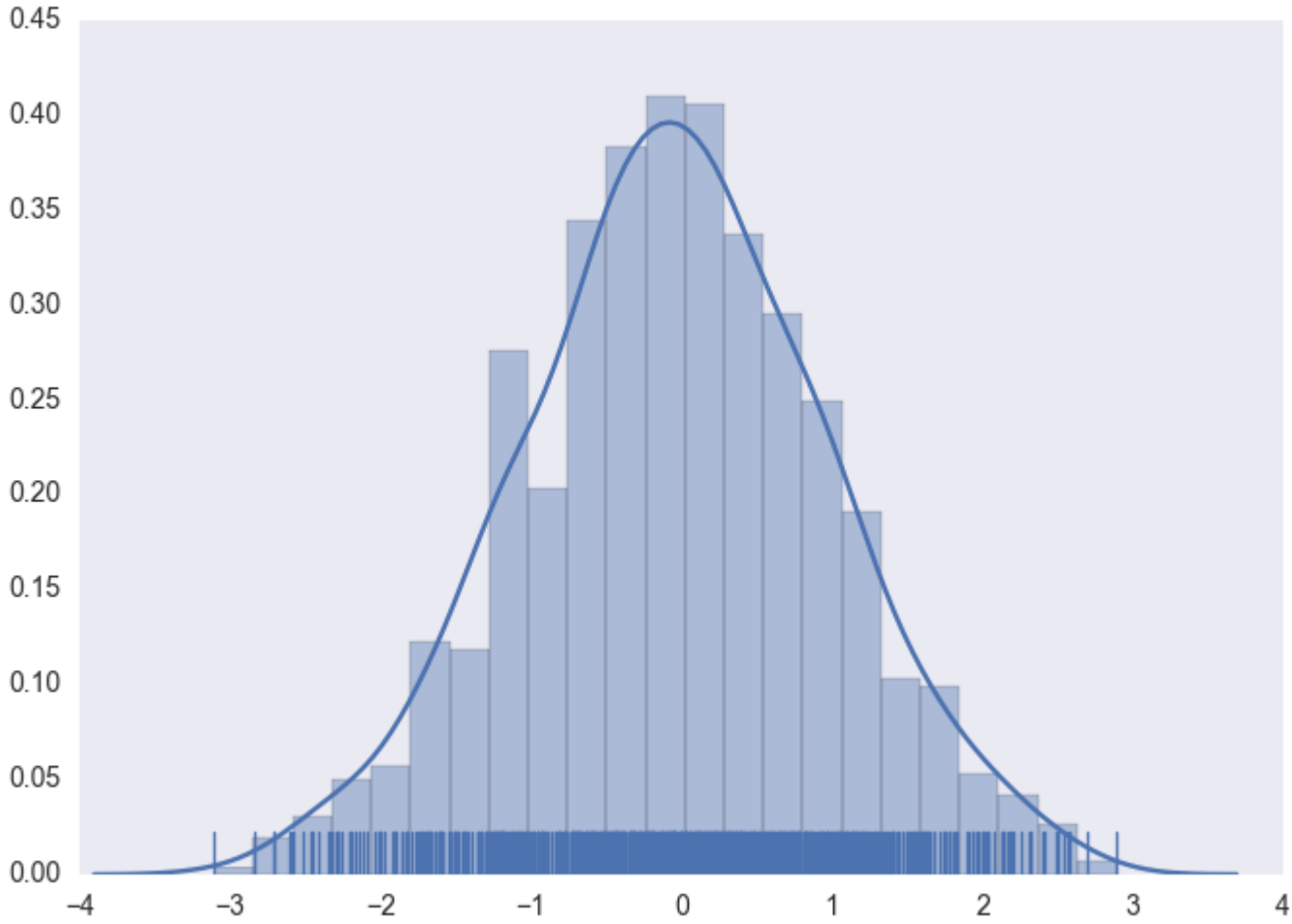
```
import numpy as np  # numpy used to create data from plotting
import seaborn as sns  # common form of importing seaborn
```

```
# Generate normally distributed data
data = np.random.randn(1000)

# Plot a histogram with both a rugplot and kde graph superimposed
sns.distplot(data, kde=True, rug=True)
```



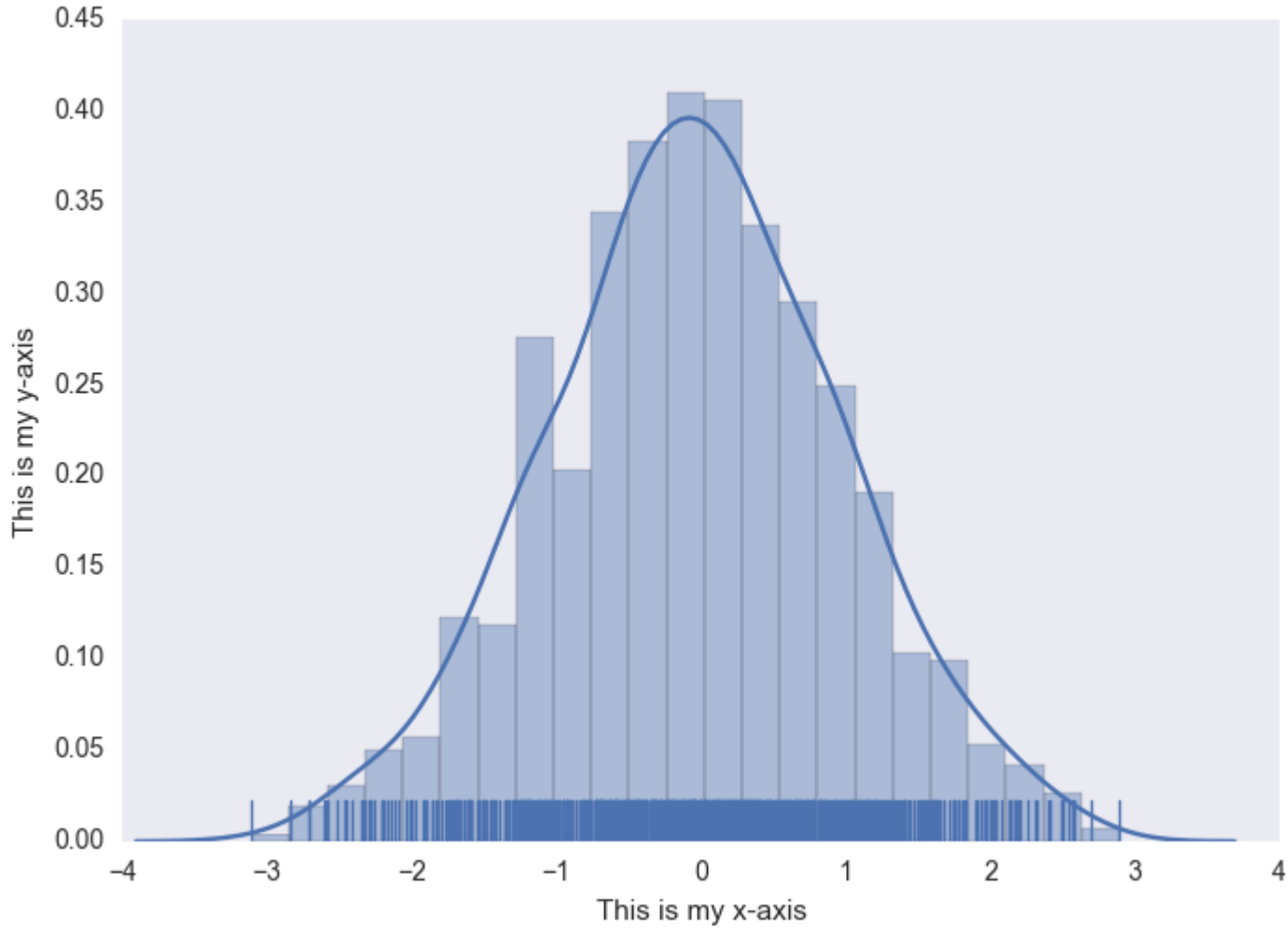
El estilo de la trama también se puede controlar mediante una sintaxis declarativa.

```
# Using previously created imports and data.

# Use a dark background with no grid.
sns.set_style('dark')
# Create the plot again
sns.distplot(data, kde=True, rug=True)
```



Como un bono adicional, los comandos normales de matplotlib todavía se pueden aplicar a los gráficos de Seaborn. Aquí hay un ejemplo de cómo agregar títulos de ejes a nuestro histograma creado anteriormente.

```
# Using previously created data and style

# Access to matplotlib commands
import matplotlib.pyplot as plt

# Previously created plot.
sns.distplot(data, kde=True, rug=True)
# Set the axis labels.
plt.xlabel('This is my x-axis')
plt.ylabel('This is my y-axis')
```



MayaVI

MayaVI es una herramienta de visualización 3D para datos científicos. Utiliza el kit de herramientas de visualización o VTK debajo del capó. Usando el poder de VTK , **MayaVI** es capaz de producir una variedad de gráficos y figuras tridimensionales. Está disponible como una aplicación de software independiente y también como una biblioteca. Similar a Matplotlib , esta biblioteca proporciona una interfaz de lenguaje de programación orientada a objetos para crear gráficos sin tener que saber acerca de **VTK** .

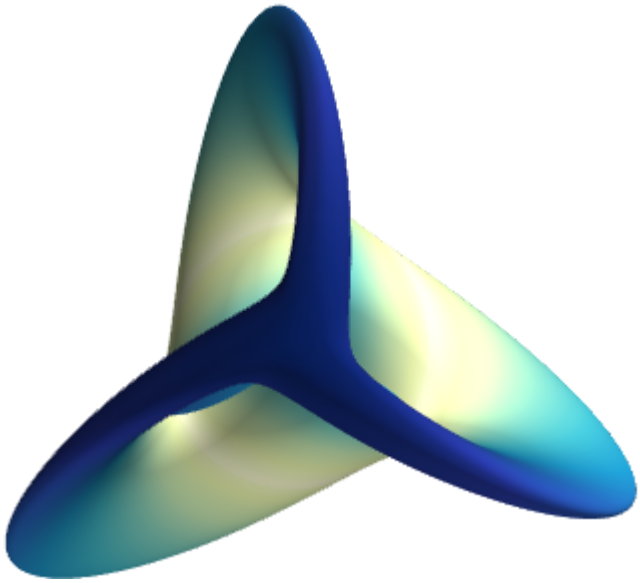
¡MayaVI está disponible solo en la serie Python 2.7x! ¡Se espera que esté disponible en la serie Python 3-x pronto! (Aunque se nota cierto éxito al usar sus dependencias en Python 3)

La documentación se puede encontrar aquí . Algunos ejemplos de galerías se encuentran aquí.

Aquí hay una muestra de la parcela creada usando **MayaVI** de la documentación.

```
# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>
# Copyright (c) 2007, Enthought, Inc.
# License: BSD Style.
```

```
from numpy import sin, cos, mgrid, pi, sqrt
from mayavi import mlab

mlab.figure(fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))
u, v = mgrid[- 0.035:pi:0.01, - 0.035:pi:0.01]

X = 2 / 3. * (cos(u) * cos(2 * v)
        + sqrt(2) * sin(u) * cos(v)) * cos(u) / (sqrt(2) -
                                                 sin(2 * u) * sin(3 * v))
Y = 2 / 3. * (cos(u) * sin(2 * v) -
        sqrt(2) * sin(u) * sin(v)) * cos(u) / (sqrt(2)
        - sin(2 * u) * sin(3 * v))
Z = -sqrt(2) * cos(u) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
S = sin(u)

mlab.mesh(X, Y, Z, scalars=S, colormap='YlGnBu', )

# Nice view from the front
mlab.view(.0, - 5.0, 4)
mlab.show()
```



Plotly

Plotly es una plataforma moderna para el trazado y visualización de datos. Útil para producir una variedad de gráficos, especialmente para ciencias de datos, **Plotly** está disponible como una biblioteca para **Python** , **R** , **JavaScript** , **Julia** y **MATLAB** . También se puede utilizar como una aplicación web con estos idiomas.

Los usuarios pueden instalar plotly library y usarlo fuera de línea después de la autenticación del usuario. La instalación de esta biblioteca y la autenticación fuera de línea se da aquí . Además, las parcelas se pueden hacer en **Jupyter Notebooks** también.

El uso de esta biblioteca requiere una cuenta con nombre de usuario y contraseña. Esto proporciona el espacio de trabajo para guardar gráficos y datos en la nube.

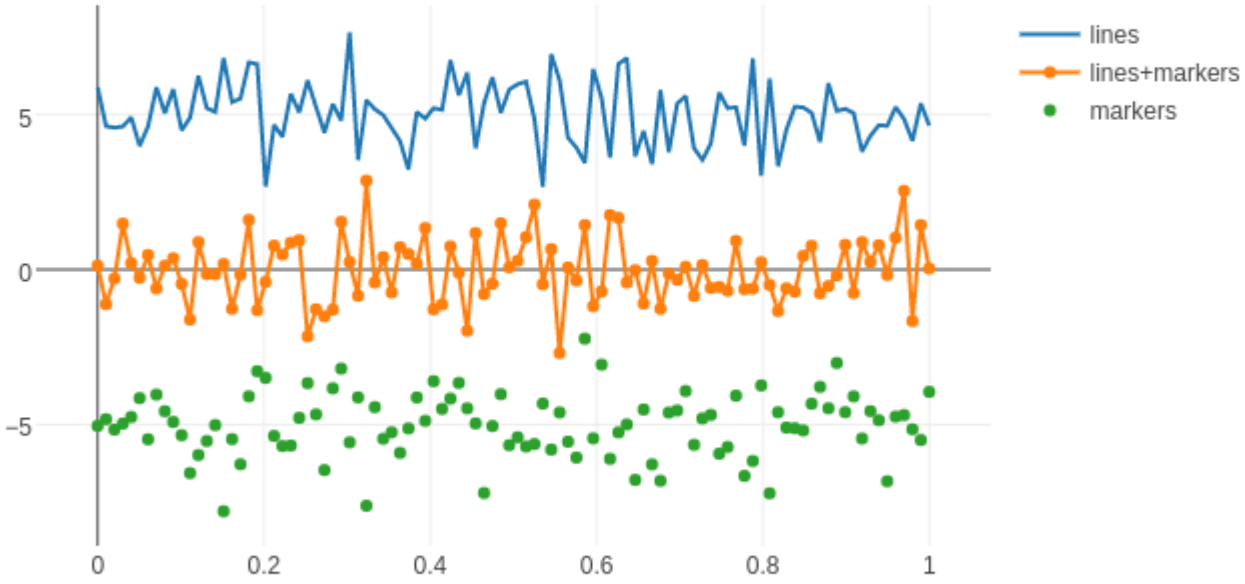
La versión gratuita de la biblioteca tiene algunas características ligeramente limitadas y está

diseñada para hacer 250 parcelas por día. La versión de pago tiene todas las características, descargas de parcelas ilimitadas y más almacenamiento de datos privados. Para más detalles, se puede visitar la página principal aquí .

Para documentación y ejemplos, se puede ir aquí.

Un diagrama de muestra de los ejemplos de documentación:

```
import plotly.graph_objs as go
import plotly as ply

# Create random data with numpy
import numpy as np

N = 100
random_x = np.linspace(0, 1, N)
random_y0 = np.random.randn(N)+5
random_y1 = np.random.randn(N)
random_y2 = np.random.randn(N)-5

# Create traces
trace0 = go.Scatter(
    x = random_x,
y = random_y0,
mode = 'lines',
name = 'lines'
)
trace1 = go.Scatter(
    x = random_x,
    y = random_y1,
    mode = 'lines+markers',
    name = 'lines+markers'
)
trace2 = go.Scatter(
    x = random_x,
    y = random_y2,
    mode = 'markers',
    name = 'markers'
)
data = [trace0, trace1, trace2]

ply.offline.plot(data, filename='line-mode')
```



Lea Visualización de datos con Python en línea:
https://riptutorial.com/es/python/topic/2388/visualizacion-de-datos-con-python

Capítulo 205: Web raspado con Python

Introducción

El raspado web es un proceso automatizado y programático a través del cual los datos se pueden " raspar " constantemente de las páginas web. También conocido como raspado de pantalla o recolección web, el raspado web puede proporcionar datos instantáneos desde cualquier página web de acceso público. En algunos sitios web, el raspado web puede ser ilegal.

Observaciones

Paquetes de Python útiles para raspado web (orden alfabético)

Realización de solicitudes y recogida de datos.

`requests`

Un paquete simple, pero poderoso para hacer peticiones HTTP.

`requests-cache`

Caché para `requests` ; almacenar datos en caché es muy útil. En desarrollo, significa que puede evitar golpear un sitio innecesariamente. Mientras ejecuta una colección real, significa que si su raspador se bloquea por algún motivo (tal vez no haya manejado algún contenido inusual en el sitio ... ¿Tal vez el sitio se haya caído ...?) Puede repetir la colección muy rápidamente de donde lo dejaste.

`scrapy`

Útil para crear rastreadores web, donde necesita algo más potente que usar `requests` e iterar a través de páginas.

`selenium`

Enlaces Python para Selenium WebDriver, para la automatización del navegador. El uso de `requests` para realizar solicitudes HTTP directamente es a menudo más sencillo para recuperar páginas web. Sin embargo, esto sigue siendo una herramienta útil cuando no es posible replicar el comportamiento deseado de un sitio usando solo las `requests` , particularmente cuando se requiere JavaScript para representar elementos en una página.

Análisis de HTML

BeautifulSoup

Consulte documentos HTML y XML, utilizando varios analizadores diferentes (el analizador HTML incorporado de Python, `html5lib` , `lxml` o `lxml.html`)

lxml

Procesos HTML y XML. Puede usarse para consultar y seleccionar contenido de documentos HTML a través de selectores de CSS y XPath.

Examples

Ejemplo básico de uso de solicitudes y lxml para raspar algunos datos

```
# For Python 2 compatibility.
from __future__ import print_function

import lxml.html
import requests


def main():
    r = requests.get("https://httpbin.org")
    html_source = r.text
    root_element = lxml.html.fromstring(html_source)
    # Note root_element.xpath() gives a *list* of results.
    # XPath specifies a path to the element we want.
    page_title = root_element.xpath('/html/head/title/text()')[0]
    print(page_title)

if __name__ == '__main__':
    main()
```

Mantenimiento de sesión web-scraping con peticiones.

Es una buena idea mantener una sesión de raspado web para conservar las cookies y otros parámetros. Además, puede dar lugar a una *mejora del rendimiento* porque `requests.Session` reutiliza la conexión TCP subyacente a un host:

```
import requests

with requests.Session() as session:
    # all requests through session now have User-Agent header set
    session.headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36'}

    # set cookies
    session.get('http://httpbin.org/cookies/set?key=value')

    # get cookies
    response = session.get('http://httpbin.org/cookies')
    print(response.text)
```

Raspado utilizando el marco de Scrapy

Primero tienes que configurar un nuevo proyecto Scrapy. Ingrese un directorio donde le gustaría almacenar su código y ejecute:

```
scrapy startproject projectName
```

Para raspar necesitamos una araña. Las arañas definen cómo se raspará un determinado sitio. Aquí está el código para una araña que sigue los enlaces a las preguntas más votadas en StackOverflow y raspa algunos datos de cada página (fuente):

```
import scrapy

class StackOverflowSpider(scrapy.Spider):
    name = 'stackoverflow'  # each spider has a unique name
    start_urls = ['http://stackoverflow.com/questions?sort=votes']  # the parsing starts from
a specific set of urls

    def parse(self, response):  # for each request this generator yields, its response is sent
to parse_question
        for href in response.css('.question-summary h3 a::attr(href)'):  # do some scraping
stuff using css selectors to find question urls
            full_url = response.urljoin(href.extract())
            yield scrapy.Request(full_url, callback=self.parse_question)

    def parse_question(self, response):
        yield {
            'title': response.css('h1 a::text').extract_first(),
            'votes': response.css('.question .vote-count-post::text').extract_first(),
            'body': response.css('.question .post-text').extract_first(),
            'tags': response.css('.question .post-tag::text').extract(),
            'link': response.url,
        }
```

Guarda tus clases de arañas en el directorio nombre de `projectName\spiders` . En este caso, `projectName\spiders\stackoverflow_spider.py` .

Ahora puedes usar tu araña. Por ejemplo, intente ejecutar (en el directorio del proyecto):

```
scrapy crawl stackoverflow
```

Modificar agente de usuario de Scrapy

En ocasiones, el host bloquea el agente de usuario de Scrapy predeterminado (`"Scrapy/VERSION (+http://scrapy.org)"`). Para cambiar el agente de usuario predeterminado, abra **settings.py** , elimine el comentario y edite la siguiente línea como desee.

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

Por ejemplo

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/51.0.2704.103 Safari/537.36'
```

Raspado utilizando BeautifulSoup4

```
from bs4 import BeautifulSoup
import requests

# Use the requests module to obtain a page
res = requests.get('https://www.codechef.com/problems/easy')

# Create a BeautifulSoup object
page = BeautifulSoup(res.text, 'lxml')   # the text field contains the source of the page

# Now use a CSS selector in order to get the table containing the list of problems
datatable_tags = page.select('table.dataTable')  # The problems are in the <table> tag,
                                                 # with class "dataTable"
# We extract the first tag from the list, since that's what we desire
datatable = datatable_tags[0]
# Now since we want problem names, they are contained in <b> tags, which are
# directly nested under <a> tags
prob_tags = datatable.select('a > b')
prob_names = [tag.getText().strip() for tag in prob_tags]

print prob_names
```

Raspado utilizando Selenium WebDriver

Algunos sitios web no les gusta ser raspado. En estos casos, es posible que necesite simular a un usuario real que trabaja con un navegador. Selenium lanza y controla un navegador web.

```
from selenium import webdriver

browser = webdriver.Firefox()  # launch firefox browser

browser.get('http://stackoverflow.com/questions?sort=votes')  # load url

title = browser.find_element_by_css_selector('h1').text  # page title (first h1 element)

questions = browser.find_elements_by_css_selector('.question-summary')  # question list

for question in questions:  # iterate over questions
    question_title = question.find_element_by_css_selector('.summary h3 a').text
    question_excerpt = question.find_element_by_css_selector('.summary .excerpt').text
    question_vote = question.find_element_by_css_selector('.stats .vote .votes .vote-count-
post').text

    print "%s\n%s\n%s votes\n-----------\n" % (question_title, question_excerpt,
question_vote)
```

El selenio puede hacer mucho más. Puede modificar las cookies del navegador, rellenar formularios, simular clics del ratón, tomar capturas de pantalla de páginas web y ejecutar JavaScript personalizado.

Descarga de contenido web simple con urllib.request

El módulo de biblioteca estándar `urllib.request` se puede usar para descargar contenido web:

```
from urllib.request import urlopen

response = urlopen('http://stackoverflow.com/questions?sort=votes')
data = response.read()

# The received bytes should usually be decoded according the response's character set
encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Un módulo similar también está disponible en Python 2 .

Raspado con rizo

importaciones:

```
from subprocess import Popen, PIPE
from lxml import etree
from io import StringIO
```

Descargando:

```
user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.95 Safari/537.36'
url = 'http://stackoverflow.com'
get = Popen(['curl', '-s', '-A', user_agent, url], stdout=PIPE)
result = get.stdout.read().decode('utf8')
```

`-s` : descarga silenciosa

`-A` : bandera de agente de usuario

Análisis:

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

Lea Web raspado con Python en línea: https://riptutorial.com/es/python/topic/1792/web-raspado-con-python

Capítulo 206: Websockets

Examples

Eco simple con aiohttp

`aiohttp` proporciona websockets asíncronos.

Python 3.x 3.5

```
import asyncio
from aiohttp import ClientSession

with ClientSession() as session:
    async def hello_world():

        websocket = await session.ws_connect("wss://echo.websocket.org")

        websocket.send_str("Hello, world!")

        print("Received:", (await websocket.receive()).data)

        await websocket.close()

    loop = asyncio.get_event_loop()
    loop.run_until_complete(hello_world())
```

Clase de envoltura con aiohttp

`aiohttp.ClientSession` puede usarse como padre para una clase de WebSocket personalizada.

Python 3.x 3.5

```
import asyncio
from aiohttp import ClientSession

class EchoWebSocket(ClientSession):

    URL = "wss://echo.websocket.org"

    def __init__(self):
        super().__init__()
        self.websocket = None

    async def connect(self):
        """Connect to the WebSocket."""
        self.websocket = await self.ws_connect(self.URL)

    async def send(self, message):
        """Send a message to the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        self.websocket.send_str(message)
        print("Sent:", message)
```

```
    async def receive(self):
        """Receive one message from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        return (await self.websocket.receive()).data

    async def read(self):
        """Read messages from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"

        while self.websocket.receive():
            message = await self.receive()
            print("Received:", message)
            if message == "Echo 9!":
                break

async def send(websocket):
    for n in range(10):
        await websocket.send("Echo {}!".format(n))
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()

with EchoWebSocket() as websocket:

    loop.run_until_complete(websocket.connect())

    tasks = (
        send(websocket),
        websocket.read()
    )

    loop.run_until_complete(asyncio.wait(tasks))

    loop.close()
```

Usando Autobahn como una Websocket Factory

El paquete Autobahn se puede utilizar para las fábricas de servidores de socket web de Python.

Documentación del paquete Python Autobahn

Para instalar, normalmente uno simplemente usaría el comando de terminal

(Para Linux):

```
sudo pip install autobahn
```

(Para ventanas):

```
python -m pip install autobahn
```

Luego, se puede crear un servidor de eco simple en un script de Python:

```
from autobahn.asyncio.websocket import WebSocketServerProtocol
class MyServerProtocol(WebSocketServerProtocol):
```

```
    '''When creating server protocol, the
    user defined class inheriting the
    WebSocketServerProtocol needs to override
    the onMessage, onConnect, et-c events for
    user specified functionality, these events
    define your server's protocol, in essence'''
    def onMessage(self,payload,isBinary):
        '''The onMessage routine is called
        when the server receives a message.
        It has the required arguments payload
        and the bool isBinary. The payload is the
        actual contents of the "message" and isBinary
        is simply a flag to let the user know that
        the payload contains binary data. I typically
        elsewise assume that the payload is a string.
        In this example, the payload is returned to sender verbatim.'''
        self.sendMessage(payload,isBinary)
if__name__=='__main__':
    try:
        importasyncio
    except ImportError:
        '''Trollius = 0.3 was renamed'''
        import trollius as asyncio
    from autobahn.asyncio.websocketimportWebSocketServerFactory
    factory=WebSocketServerFactory()
    '''Initialize the websocket factory, and set the protocol to the
    above defined protocol(the class that inherits from
    autobahn.asyncio.websocket.WebSocketServerProtocol)'''
    factory.protocol=MyServerProtocol
    '''This above line can be thought of as "binding" the methods
    onConnect, onMessage, et-c that were described in the MyServerProtocol class
    to the server, setting the servers functionality, ie, protocol'''
    loop=asyncio.get_event_loop()
    coro=loop.create_server(factory,'127.0.0.1',9000)
    server=loop.run_until_complete(coro)
    '''Run the server in an infinite loop'''
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.close()
        loop.close()
```

En este ejemplo, se está creando un servidor en el host local (127.0.0.1) en el puerto 9000. Esta es la IP y el puerto de escucha. Esta información es importante, ya que al usar esto, puede identificar la dirección LAN de su computadora y el puerto hacia adelante desde su módem, aunque cualquier enrutador que tenga a la computadora. Luego, utilizando google para investigar su IP WAN, puede diseñar su sitio web para enviar mensajes WebSocket a su IP WAN, en el puerto 9000 (en este ejemplo).

Es importante que retroceda desde su módem, lo que significa que si tiene enrutadores conectados en cadena al módem, ingrese a los ajustes de configuración del módem, transfiera el puerto desde el módem al enrutador conectado, y así sucesivamente hasta el enrutador final de su computadora está conectado está recibiendo la información que se está recibiendo en el puerto de módem 9000 (en este ejemplo).

Lea Websockets en línea: https://riptutorial.com/es/python/topic/4751/websockets

Creditos

S. No	Capítulos	Contributors
1	Empezando con Python Language	A. Raza, Aaron Critchley, Abhishek Jain, AER, afeique, Akshay Kathpal, alejosocorro, Alessandro Trinca Tornidor, Alex Logan, ALinuxLover, Andrea, Andrii Abramov, Andy, Andy Hayden, angussidney, Ani Menon, Anthony Pham, Antoine Bolvy, Aquib Javed Khan, Ares, Arpit Solanki, B8vrede, Baaing Cow, baranskistad, Brian C, Bryan P, BSL-5, BusyAnt, Cbeb24404, ceruleus, ChaoticTwist, Charlie H, Chris Midgley, Christian Ternus, Claudiu, Clíodhna, CodenameLambda, cLDs EED, Community, Conrad.Dean, Daksh Gupta, Dania, Daniel Minnaar, Darth Shadow, Dartmouth, deeenes, Delgan, depperm, DevD, dodell, Douglas Starnes, duckman_1991, Eamon Charles, edawine, Elazar, eli-bd, Enrico Maria De Angelis, Erica, Erica, ericdwang, Erik Godard, EsmaeelE, Filip Haglund, Firix, fox, Franck Dernoncourt, Fred Barclay, Freddy, Gerard Roche, glS, GoatsWearHats, GThamizh, H. Pauwelyn, hardmooth, hayalci, hichris123, Ian, IanAuld, icesin, Igor Raush, Ilyas Mimouni, itsthejoker, J F, Jabba, jalanb, James, James Taylor, Jean-Francois T., jedwards, Jeffrey Lin, jfunez, JGreenwell, Jim Fasarakis Hilliard, jim opleydulven, jimsug, jmunsch, Johan Lundberg, John Donner, John Slegers, john400, jonrsharpe, Joseph True, JRodDynamite, jtbandes, Juan T, Kamran Mackey, Karan Chudasama, KerDam, Kevin Brown, Kiran Vemuri, kisanme, Lafexlos, Leon, Leszek Kicior, LostAvatar, Majid, manu, MANU, Mark Miller, Martijn Pieters, Mathias711, matsjoyce, Matt, Mattew Whitt, mdegis, Mechanic, Media, mertyildiran, metahost, Mike Driscoll, MikJR, Miljen Mikic, mnoronha, Morgoth, moshemeirelles, MSD, MSeifert, msohng, msw, muddyfish, Mukund B, Muntasir Alam, Nathan Arthur, Nathaniel Ford, Ned Batchelder, Ni., niyasc, noɥʎd ʎzɐɹƆ, numbermaniac, orvi, Panda, Patrick Haugh, Pavan Nath, Peter Masiar, PSN, PsyKzz, pylang, pzp, Qchmqs, Quill, Rahul Nair, Rakitić, Ram Grandhi, rfkortekaas, rick112358, Robotski, rrao, Ryan Hilbert, Sam Krygsheld, Sangeeth Sudheer, SashaZd, Selcuk, Severiano Jaramillo Quintanar, Shiven, Shoe, Shog9, Sigitas Mockus, Simplans, Slayther, stark, StuxCrystal, SuperBiasedMan, Sнађошƒа, taylor swift, techydesigner, Tejus Prasad, TerryA, The_Curry_Man, TheGenie OfTruth, Timotheus.Kampik, tjohnson, Tom Barron, Tom de Geus, Tony Suffolk 66, tonyo, TPVasconcelos,

		user2314737, user2853437, user312016, Utsav T, vaichidrewar, vasili111, Vin, W.Wong, weewooquestionaire, Will, wintermute, Yogendra Sharma, Zach Janicki, Zags
2	* args y ** kwargs	cjds, Eric Zhang, ericmarkmartin, Geeklhem, J F, Jeff Hutchins, Jim Fasarakis Hilliard, JuanPablo, kdopen, loading..., Marlon Abeykoon, Mattew Whitt, Pasha, pcurry, PsyKzz, Scott Mermelstein, user2314737, Valentin Lorentz, Veedrac
3	Acceso a la base de datos	Alessandro Trinca Tornidor, Antonio, bee-sting, cldseed, D. Alveno, John Y, LostAvatar, mbsingh, Michel Touw, qwertyuip9, RamenChef, rrawat, Stephen Leppik, Stephen Nyamweya, sumitroy, user2314737, valeas, zweiterlinde
4	Acceso al código fuente y código de bytes de Python	muddyfish, StuxCrystal, user2314737
5	Acceso de atributo	Elazar, SashaZd, SuperBiasedMan
6	agrupar por()	Parousia, Thomas Gerot
7	Alcance variable y vinculante	Anthony Pham, davidism, Elazar, Esteis, Mike Driscoll, SuperBiasedMan, user2314737, zvone
8	Almohada	Razik
9	Alternativas para cambiar la declaración de otros idiomas	davidism, J F, zmo, Валерий Павлов
10	Ambiente Virtual Python - virtualenv	Vikash Kumar Jain
11	Análisis de argumentos de línea de comandos	amblina, Braiam, Claudiu, cledoux, Elazar, Gerard Roche, krato, loading..., Marco Pashkov, Or Duan, Pasha, RamenChef, rfkortekaas, Simplans, Thomas Gerot, Topperfalkon, zmo, zondo
12	Análisis de HTML	alecxe, talhasch
13	Anti-patrones de Python	Alessandro Trinca Tornidor, Annonymous, eenblam, Mahmoud Hashemi, RamenChef, Stephen Leppik
14	Apilar	ADITYA, boboquack, Chromium, cjds, depperm, Hannes Karppila, JGreenwell, Jonatan, kdopen, OliPro007, orvi, SashaZd, Снађошfа, textshell, Thomas Ahle, user2314737
15	Árbol de sintaxis	Teepeemm

	abstracta	
16	Archivos y carpetas I / O	Ajean, Anthony Pham, avb, Benjamin Hodgson, Bharel, Charles, crhodes, David Cullen, Dov, Esteis, ilse2005, isvforall, jfsturtz, Justin, Kevin Brown, mattgathu, MSeifert, nlsdfnbch, Ozair Kafray, PYPL, pzp, RamenChef, Ronen Ness, rrao, Serenity, Simplans, SuperBiasedMan, Tasdik Rahman, Thomas Gerot, Umibozu, user2314737, Will, WombatPM, xgord
17	ArcPy	Midavalo, PolyGeo, Zhanping Shi
18	Arrays	Andy, Pavan Nath, RamenChef, Vin
19	Audio	blueberryfields, Comrade SparklePony, frankyjuang, jmunsch, orvi, qwertyuip9, Stephen Leppik, Thomas Gerot
20	Aumentar errores / excepciones personalizados	naren
21	Biblioteca de subproceso	Adam Matan, Andrew Schade, Brendan Abel, jfs, jmunsch, Riccardo Petraglia
22	Bloques de código, marcos de ejecución y espacios de nombres.	Jeremy, Mohammed Salman
23	Bucles	Adriano, Alex L, alfonso.kim, Alleo, Anthony Pham, Antti Haapala, Chris Hunt, Christian Ternus, Darth Kotik, DeepSpace, Delgan, DhiaTN, ebo, Elazar, Eric Finn, Felix D., Ffisegydd, Gal Dreiman, Generic Snake, ghostarbeiter, GoatsWearHats, Guy, Inbar Rose, intboolstring, J F, James, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, jrast, Karl Knechtel, machine yearning, Mahdi, manetsus, Martijn Pieters, Math, Mathias711, MSeifert, pnhgiol, rajah9, Rishabh Gupta, Ryan, sarvajeetsuman, sevenforce, SiggyF, Simplans, skrrgwasme, SuperBiasedMan, textshell, The_Curry_Man, Thomas Gerot, Tom, Tony Suffolk 66, user1349663, user2314737, Vinzee, Will
24	buscando	Dan Sanderson, Igor Raush, MSeifert
25	Características ocultas	Aaron Hall, Akshat Mahajan, Anthony Pham, Antti Haapala, Byte Commander, dermen, Elazar, Ellis, ericmarkmartin, Fermi paradox, Ffisegydd, japborst, Jim Fasarakis Hilliard, jonrsharpe, Justin, kramer65, Lafexlos, LDP, Morgan Thrapp, muddyfish, nico, OrangeTux, pcurry, Pythonista, Selcuk, Serenity, Tejas

		Jadhav, tobias_k, Vlad Shcherbina, Will
26	ChemPy - paquete de python	Biswa_9937
27	Clases base abstractas (abc)	Akshat Mahajan, Alessandro Trinca Tornidor, JGreenwell, Kevin Brown, Mattew Whitt, mkrieger1, SashaZd, Stephen Leppik
28	Clasificación, mínimo y máximo	Antti Haapala, APerson, GoatsWearHats, Mirec Miskuf, MSeifert, RamenChef, Simplans, Valentin Lorentz
29	Comentarios y Documentación	Ani Menon, FunkySayu, MattCorr, SuperBiasedMan, TuringTux
30	Comparaciones	Anthony Pham, Ares, Elazar, J F, MSeifert, Shawn Mehan, SuperBiasedMan, Will, Xavier Combelle
31	Complementos y clases de extensión	2Cubed, proprefenetre, pylang, rrao, Simon Hibbs, Simplans
32	Comprobando la existencia de ruta y permisos	Esteis, Marlon Abeykoon, mnoronha, PYPL
33	Computación paralela	Akshat Mahajan, Dair, Franck Dernoncourt, J F, Mahdi, nlsdfnbch, Ryan Smith, Vinzee, Xavier Combelle
34	Comunicación Serial Python (pyserial)	Alessandro Trinca Tornidor, Ani Menon, girish946, mnoronha, Saranjith, user2314737
35	Concurrencia de Python	David Heyman, Faiz Halde, Iván Rodríguez Torres, J F, Thomas Moreau, Tyler Gubala
36	Condicionales	Andy Hayden, BusyAnt, Chris Larson, deepakkt, Delgan, Elazar, evuez, Ffisegydd, Geeklhem, Hannes Karppila, James, Kevin Brown, krato, Max Feng, noɥʎɹɐzʎɹɐƆ, rajah9, rrao, SashaZd, Simplans, Slayther, Soumendra Kumar Sahoo, Thomas Gerot, Trimax, Valentin Lorentz, Vinzee, wwii, xgord, Zack
37	Conectando Python a SQL Server	metmirr
38	Conexión segura de shell en Python	mnoronha, Shijo
39	configparser	Chinmay Hegde, Dunatotatos
40	Conjunto	Andrzej Pronobis, Andy Hayden, Bahrom, Cimbali, Cody

		Piersall, Conrad.Dean, Elazar, evuez, J F, James, Or East, pylang, RahulHP, RamenChef, Simplans, user2314737
41	Contando	Andy Hayden, MSeifert, Peter Mølgaard Pallesen, pylang
42	Copiando datos	hashcode55, StuxCrystal
43	Corte de listas (selección de partes de listas)	Greg, JakeD
44	Creando paquetes de Python	Claudiu, KeyWeeUsr, Marco Pashkov, pylang, SuperBiasedMan, Thtu
45	Creando un servicio de Windows usando Python	Simon Hibbs
46	Crear entorno virtual con virtualenvwrapper en windows	Sirajus Salayhin
47	ctypes	Or East
48	Datos binarios	Eleftheria, evuez, mnoronha
49	Decoradores	Alessandro Trinca Tornidor, ChaoticTwist, Community, Dair, doratheexplorer0911, Emolga, greut, iankit, JGreenwell, jonrsharpe, kefkius, Kevin Brown, Mattew Whitt, MSeifert, muddyfish, Mukunda Modell, Nearoo, Nemo, Nuno André, Pasha, Rob Bednark, seenu s, Shreyash S Sarnayak, Simplans, StuxCrystal, Suhas K, technusm1, Thomas Gerot, tyteen4a03, Wladimir Palant, zvone
50	Definiendo funciones con argumentos de lista	zenlc2000
51	dejar de lado	Biswa_9937
52	Depuración	Aldo, B8vrede, joel3000, Sardathrion, Sardorbek Imomaliev, Vlad Bezden
53	Descomprimir archivos	andrew
54	Descriptor	bbayles, cizixs, Nemo, pylang, SuperBiasedMan
55	Despliegue	Gal Dreiman, Iancnorden, Wayne Werner

56	Diccionario	Amir Rachum, Anthony Pham, APerson, ArtOfCode, BoppreH, Burhan Khalid, Chris Mueller, cizixs, depperm, Ffisegydd, Gareth Latty, Guy, helpful, iBelieve, Igor Raush, Infinity, James , JGreenwell, jonrsharpe, Karsten 7., kdopen, machine yearning, Majid, mattgathu, Mechanic, MSeifert, muddyfish, Nathan, nlsdfnbch, noɥʎdʎzɐɔ, ronrest, Roy Iacob, Shawn Mehan, Simplans, SuperBiasedMan, TehTris, Valentin Lorentz, viveksyngh, Xavier Combelle
57	Diferencia entre Módulo y Paquete	DeepSpace, Simplans, tjohnson
58	Distribución	Alessandro Trinca Tornidor, JGreenwell, metahost, Pigman168 , RamenChef, Stephen Leppik
59	Django	code_geek, orvi
60	Ejecución de código dinámico con `exec` y` eval`	Antti Haapala, Ilja Everilä
61	El dis módulo	muddyfish, user2314737
62	El intérprete (consola de línea de comandos)	Aaron Christiansen, David, Elazar, Peter Shinners, ppperry
63	El módulo base64	Thomas Gerot
64	El módulo de configuración regional	Will, XonAether
65	El módulo os	Andy, Christian Ternus, JelmerS, JL Peyret, mnoronha, Vinzee
66	Empezando con GZip	orvi
67	Enchufes	David Cullen, Dev, MattCorr, nlsdfnbch, Rob H, StuxCrystal, textshell, Thomas Gerot, Will
68	entorno virtual con virtualenvwrapper	Sirajus Salayhin
69	Entornos virtuales	Adrian17, Artem Kolontay, ArtOfCode, Bhargav, brennan, Dair, Daniil Ryzhkov, Darkade, Darth Shadow, edwinksl, Fernando, ghostarbeiter, ha_1694, Hans Then, Iancnorden, J F, Majid, Marco Pashkov, Matt Giltaji, Mattew Whitt, nehemiah, Nuhil Mehdy, Ortomala Lokni, Preston, pylang, qwertyuip9,

		RamenChef, Régis B., Sebastian Schrader, Serenity, Shantanu Alshi, Shrey Gupta, Simon Fraser, Simplans, wrwrwr , ychaouche, zopieux, zvezda
70	Entrada y salida básica	Doraemon, GoatsWearHats, J F, JNat, Marco Pashkov, Mark Miller, Martijn Pieters, Nathaniel Ford, Nicolás, pcurry, pzp, SashaZd, SuperBiasedMan, Vilmar
71	Entrada, subconjunto y salida de archivos de datos externos utilizando Pandas	Mark Miller
72	Enumerar	Andy, Elazar, evuez, Martijn Pieters, techydesigner
73	Errores comunes	abukaj, ADITYA, Alec, Alessandro Trinca Tornidor, Alex, Antoine Bolvy, Baaing Cow, Bhargav Rao, Billy, bixel, Charles, Cheney, Christophe Roussy, Dartmouth, DeepSpace, DhiaTN, Dilettant, fox, Fred Barclay, Gerard Roche, greatwolf, hiro protagonist, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, Lafexlos, maazza, Malt, Mark, matsjoyce, Matt Dodge, MervS, MSeifert, ncmathsadist, omgimanerd, Patrick Haugh, pylang, RamenChef, Reut Sharabani, Rob Bednark, rrao, SashaZd, Shihab Shahriar, Simplans, SuperBiasedMan, Tim D, Tom Dunbavan, tyteen4a03, user2314737, Will Vousden, Wombatz
74	Escribiendo a CSV desde String o List	Hriddhi Dey, Thomas Crowley
75	Eventos enviados de Python Server	Nick Humrich
76	Examen de la unidad	Alireza Savand, Ami Tavory, antimatter15, Arpit Solanki, bijancn, Claudiu, Dartmouth, engineercoding, Ffisegydd, J F, JGreenwell, jmunsch, joel3000, Kevin Brown, Kinifwyne, Mario Corchero, Matt Giltaji, Mattew Whitt, mgilson, muddyfish, pylang, strpeter
77	Excepciones	Adrian Antunez, Alessandro Trinca Tornidor, Alfe, Andy, Benjamin Hodgson, Brian Rodriguez, BusyAnt, Claudiu, driax, Elazar, flazzarini, ghostarbeiter, Ilia Barahovski, J F, Marco Pashkov, muddyfish, noɥʎdʎzɐɔƆ, Paul Weaver, Rahul Nair, RamenChef, Shawn Mehan, Shiven, Shkelqim Memolla, Simplans, Slickytail, Stephen Leppik, Sudip Bhandari, SuperBiasedMan, user2314737
78	Excepciones del Commonwealth	Juan T, TemporalWolf

79	Explotación florestal	Gal Dreiman, Jörn Hees, sxnwlfkk
80	Exposiciónción	Anthony Pham, intboolstring, jtbandes, Luke Taylor, MSeifert, Pasha, supersam654
81	Expresiones Regulares (Regex)	Aidan, alejosocorro, andandandand, Andy Hayden, ashes999, B8vrede, Claudiu, Darth Shadow, driax, Fermi paradox, ganesh gadila, goodmami, Jan, Jeffrey Lin, jonrsharpe, Julien Spronck, Kevin Brown, Md.Sifatul Islam, Michael M., mnoronha , Nander Speerstra, nrusch, Or East, orvi, regnarg, sarvajeetsuman, Simplans, SN Ravichandran KR, SuperBiasedMan, user2314737, zondo
82	Extensiones de escritura	Dartmouth, J F, mattgathu, Nathan Osman, techydesigner, ygram
83	Fecha y hora	Ajean, alecxe, Andy, Antti Haapala, BusyAnt, Conrad.Dean, Elazar, ghostarbeiter, J F, Jeffrey Lin, jonrsharpe, Kevin Brown , Nicole White, nlsdfnbch, Ohad Eytan, Paul, paulmorriss, proprius, RahulHP, RamenChef, sagism, Simplans, Sirajus Salayhin, Suku, Will
84	Filtrar	APerson, cfi, J Atkin, MSeifert, rajah9, SuperBiasedMan
85	Formato de cadena	4444, Aaron Christiansen, Adam_92, ADITYA, Akshit Soota, aldanor, alecxe, Alessandro Trinca Tornidor, Andy Hayden, Ani Menon, B8vrede, Bahrom, Bhargav, Charles, Chris, Darth Shadow, Dartmouth, Dave J, Delgan, dreftymac, evuez, Franck Dernoncourt, Gal Dreiman, gerrit, Giannis Spiliopoulos, GiantsLoveDeathMetal, goyalankit, Harrison, James Elderfield, Jean-Francois T., Jeffrey Lin, jetpack_guy, JL Peyret, joel3000, Jonatan, JRodDynamite, Justin, Kevin Brown, knight, krato, Marco Pashkov, Mark, Matt, Matt Giltaji, mu , MYGz, Nander Speerstra, Nathan Arthur, Nour Chawich, orion_tvv, ragesz, SashaZd, Serenity, serv-inc, Simplans, Slayther, Sometowngeek, SuperBiasedMan, Thomas Gerot, tobias_k, Tony Suffolk 66, UloPe, user2314737, user312016, Vin, zondo
86	Formato de fecha	surfthecity
87	Función de mapa	APerson, cfi, Igor Raush, Jon Ericson, Karl Knechtel, Marco Pashkov, MSeifert, noɥʞɹʎzɐɔ, Parousia, Simplans, SuperBiasedMan, tlama, user2314737
88	Funciones	Adriano, Akshat Mahajan, AlexV, Andy, Andy Hayden, Anthony Pham, Arkady, B8vrede, Benjamin Hodgson, btel, CamelBackNotation, Camsbury, Chandan Purohit, ChaoticTwist, Charlie H, Chris Larson, Community, D. Alveno, danidee, DawnPaladin, Delgan, duan, duckman_1991, elegent

		, Elodin, Emma, EsmaeelE, Ffisegydd, Gal Dreiman, ghostarbeiter, Hurkyl, J F, James, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, jkitchen, Jossie Calderon, Justin, Kevin Brown, L3viathan, Lee Netherton, Martijn Pieters, Martin Thurau, Matt Giltaji, Mike - SMT, Mike Driscoll, MSeifert, muddyfish, Murphy4, nd., noɥʎdʎzɐɹƆ, Pasha, pylang, pzp, Rahul Nair, Severiano Jaramillo Quintanar, Simplans, Slayther, Steve Barnes, Steven Maude, SuperBiasedMan, textshell, then0rTh, Thomas Gerot, user2314737, user3333708, user405 , Utsav T, vaultah, Veedrac, Will, Will, zxxz, λuser
89	Funciones parciales	FrankBr
90	Generadores	2Cubed, Ahsanul Haque, Akshat Mahajan, Andy Hayden, Arthur Dent, ArtOfCode, Augustin, Barry, Chankey Pathak, Claudiu, CodenameLambda, Community, deeenes, Delgan, Devesh Saini, Elazar, ericmarkmartin, Ernir, ForceBru, Igor Raush, Ilia Barahovski, J0HN, jackskis, Jim Fasarakis Hilliard, Juan T, Julius Bullinger, Karl Knechtel, Kevin Brown, Kronen, Luc M, Lyndsy Simon, machine yearning, Martijn Pieters, Matt Giltaji, max, MSeifert, nlsdfnbch, Pasha, Pedro, PsyKzz, pzp, satsumas, sevenforce, Signal, Simplans, Slayther, StuxCrystal, tversteeg, Valentin Lorentz, Will, William Merrill, xtreak, Zaid Ajaj, zarak, λuser
91	Gestores de contexto (declaración “con”)	Abhijeet Kasurde, Alessandro Trinca Tornidor, Andy Hayden, Antoine Bolvy, carrdelling, Conrad.Dean, Dartmouth, David Marx, DeepSpace, Elazar, Kevin Brown, magu_, Majid, Martijn Pieters, Matthew, nlsdfnbch, Pasha, Peter Brittain, petrs, Shuo, Simplans, SuperBiasedMan, The_Cthulhu_Kid, Thomas Gerot, tyteen4a03, user312016, Valentin Lorentz, vaultah, λuser
92	Gráficos de tortuga	Luca Van Oort, Stephen Leppik
93	hashlib	Mark Omo, xiaoyi
94	Heapq	ettanany
95	Herramienta 2to3	Alessandro Trinca Tornidor, Dartmouth, Firix, Kevin Brown, Naga2Raja, Stephen Leppik
96	herramienta grafica	xiaoyi
97	ijson	Prem Narain
98	Implementaciones no oficiales de Python	Jacques de Hooge, Squidward
99	Importando modulos	angussidney, Anthony Pham, Antonis Kalou, Brett Cannon,

		BusyAnt, Casebash, Christian Ternus, Community, Conrad.Dean, Daniel, Dartmouth, Esteis, Ffisegydd, FMc, Gerard Roche, Gideon Buckwalter, J F, JGreenwell, Kinifwyne, languitar, Lex Scarisbrick, Matt Giltaji, MSeifert, niyasc, nlsdfnbch, Paulo Freitas, pylang, Rahul Nair, Saiful Azad, Serenity, Simplans, StardustGogeta, StuxCrystal, SuperBiasedMan, techydesigner, the_cat_lady, Thomas Gerot, Tony Meyer, Tushortz, user2683246, Valentin Lorentz, Valor Naram, vaultah, wnnmaw
100	Incompatibilidades que se mueven de Python 2 a Python 3	671620616, Abhishek Kumar, Akshit Soota, Alex Gaynor, Allan Burleson, Alleo, Amarpreet Singh, Andy Hayden, Ani Menon, Antoine Bolvy, AntsySysHack, Antti Haapala, Antwan, arekolek , Ares, asmeurer, B8vrede, Bakuriu, Bharel, Bhargav Rao, bignose, bitchaser, Bluethon, Cache Staheli, Cameron Gagnon , Charles, Charlie H, Chris Sprague, Claudiu, Clayton Wahlstrom, cʟdseed, Colin Yang, Cometsong, Community, Conrad.Dean, danidee, Daniel Stradowski, Darth Shadow, Dartmouth, Dave J, David Cullen, David Heyman, deeenes, DeepSpace, Delgan, DoHe, Duh-Wayne-101, Dunno, dwanderson, Ekeyme Mo, Elazar, enderland, enrico.bacis, erewok, ericdwang, ericmarkmartin, Ernir, ettanany, Everyone_Else, evuez, Franck Dernoncourt, Fred Barclay, garg10may, Gavin, geoffspear, ghostarbeiter, GoatsWearHats, H. Pauwelyn, Haohu Shen, holdenweb, iScrE4m, Iván C., J F, J. C. Leitão, James Elderfield, James Thiele, jarondl, jedwards, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, Jimmy Song, John Slegers, Jojodmo, jonrsharpe, Josh, Juan T, Justin, Justin M. Ucar, Kabie, kamalbanga, Karl Knechtel, Kevin Brown, King's jester, Kunal Marwaha, Lafexlos, lenz, linkdd, l'L'l, Mahdi, Martijn Pieters, Martin Thoma, masnun, Matt, Matt Dodge, Matt Rowland, Mattew Whitt, Max Feng, mgwilliams, Michael Recachinas, mkj, mnoronha, Moinuddin Quadri, muddyfish, Nathaniel Ford, niemmi, niyasc, noɥʎdʎzɐɔ, OrangeTux, Pasha, Paul Weaver, Paulo Freitas, pcurry, pktangyue, poppie, pylang, python273, Pythonista, RahulHP, Rakitić, RamenChef, Rauf, René G, rfkortekaas, rrao, Ryan, sblair, Scott Mermelstein, Selcuk, Serenity, Seth M. Larson, ShadowRanger, Simplans, Slayther, solarc, sricharan, Steven Hewitt, sth, SuperBiasedMan, Tadhg McDonald-Jensen, techydesigner, Thomas Gerot, Tim, tobias_k, Tyler, tyteen4a03 , user2314737, user312016, Valentin Lorentz, Veedrac, Ven, Vinayak, Vlad Shcherbina, VPfB, WeizhongTu, Wieland, wim, Wolf, Wombatz, xtreak, zarak, zcb, zopieux, zurfyx, zvezda
101	Indexación y corte	Alleo, amblina, Antoine Bolvy, Bonifacio2, Ffisegydd, Guy, Igor Raush, Jonatan, Martec, MSeifert, MUSR, pzp, RahulHP, Reut

		Sharabani, SashaZd, Sayed M Ahamad, SuperBiasedMan, theheadofabroom, user2314737, yurib
102	Interfaz de puerta de enlace de servidor web (WSGI)	David Heyman, Kevin Brown, Preston, techydesigner
103	Introducción a RabbitMQ utilizando AMQPStorm	eandersson
104	Iterables e iteradores	4444, Conrad.Dean, demonplus, Ilia Barahovski, Pythonista
105	kivy - Framework Python multiplataforma para el desarrollo de NUI	dhimanta
106	La declaración de pase	Anaphory
107	La función de impresión	Beall619, Frustrated, Justin, Leon Z., lukewrites, SuperBiasedMan, Valentin Lorentz
108	La variable especial __name__	Annonymous, BusyAnt, Christian Ternus, jonrsharpe, Lutz Prechelt, Steven Elliott
109	Las clases	Aaron Hall, Ahsanul Haque, Akshat Mahajan, Andrzej Pronobis , Anthony Pham, Avantol13, Camsbury, cfi, Community, Conrad.Dean, Daksh Gupta, Darth Shadow, Dartmouth, depperm, Elazar, Ffisegydd, Haris, Igor Raush, InitializeSahib, J F, jkdev, jlarsch, John Militer, Jonas S, Jonathan, Kallz, KartikKannapur, Kevin Brown, Kinifwyne, Leo, Liteye, lmiguelvargasf, Mailerdaimon, Martijn Pieters, Massimiliano Kraus, Mattew Whitt, MrP01, Nathan Arthur, ojas mohril, Pasha, Peter Steele, pistache, Preston, pylang, Richard Fitzhugh, rohittk239, Rushy Panchal, Sempoo, Simplans, Soumendra Kumar Sahoo, SuperBiasedMan, techydesigner, then0rTh, Thomas Gerot, Tony Suffolk 66, tox123, UltraBob, user2314737, wrwrwr, Yogendra Sharma
110	Lectura y Escritura CSV	Adam Matan, Franck Dernoncourt, Martin Valgur, mnoronha, ravigadila, Setu
111	Lista	Adriano, Alexander, Anthony Pham, Ares, Barry, blueenvelope, Bosoneando, BusyAnt, Çağatay Uslu, caped114, Chandan Purohit, ChaoticTwist, cizixs, Daniel Porteous, Darth Kotik, deeenes, Delgan, Elazar, Ellis, Emma, evuez, exhuma, Ffisegydd, Flickerlight, Gal Dreiman, ganesh gadila,

		ghostarbeiter, Igor Raush, intboolstring, J F, j3485, jalanb, James, James Elderfield, jani, jimsug, jkdev, JNat, jonrsharpe, KartikKannapur, Kevin Brown, Lafexlos, LDP, Leo Thumma, Luke Taylor, lukewrites, lxer, Majid, Mechanic, MrP01, MSeifert, muddyfish, n12312, noɥʎdʌzɐɔO, Oz Bar-Shalom, Pasha, Pavan Nath, poke, RamenChef, ravigadila, ronrest, Serenity, Severiano Jaramillo Quintanar, Shawn Mehan, Simplans, sirin, solarc, SuperBiasedMan, textshell, The_Cthulhu_Kid, user2314737, user6457549, Utsav T, Valentin Lorentz, vaultah, Will, wythagoras, Xavier Combelle
112	Lista de Comprensiones	3442, Akshit Soota, André Laszlo, Andy Hayden, Annonymous, Ari, Bhargav, Chris Mueller, Darth Shadow, Dartmouth, Delgan, enrico.bacis, Franck Dernoncourt, garg10may, intboolstring, Jeff Langemeier, Josh Caswell, JRodDynamite, justhalf, kdopen, Ken T, Kevin Brown, kiliantics, longyue0521, Martijn Pieters, Mattew Whitt, Moinuddin Quadri, MSeifert, muddyfish, noɥʎdʌzɐɔO, pktangyue, Pyth0nicPenguin, Rahul Nair, Riccardo Petraglia, SashaZd, shrishinde, Simplans, Slayther, sudo bangbang, theheadofabroom, then0rTh, Tim McNamara, Udi, Valentin Lorentz, Veedrac, Zags
113	Lista de desestructuración (también conocido como embalaje y desembalaje)	J F, sth, zmo
114	Listar comprensiones	3442, 4444, acdr, Ahsanul Haque, Akshay Anand, Akshit Soota, Alleo, Amir Rachum, André Laszlo, Andy Hayden, Ankit Kumar Singh, Antoine Bolvy, APerson, Ashwinee K Jha, B8vrede, bfontaine, Brian Cline, Brien, Casebash, Celeo, cfi, ChaoticTwist, Charles, Charlie H, Chong Tang, Community, Conrad.Dean, Dair, Daniel Stradowski, Darth Shadow, Dartmouth, David Heyman, Delgan, Dima Tisnek, eenblam, Elazar, Emma, enrico.bacis, EOL, ericdwang, ericmarkmartin, Esteis, Faiz Halde, Felk, Fermi paradox, Florian Bender, Franck Dernoncourt, Fred Barclay, freidrichen, G M, Gal Dreiman, garg10may, ghostarbeiter, GingerHead, griswolf, Hannele, Harry, Hurkyl, IanAuld, iankit, Infinity, intboolstring, J F, J0HN, James, JamesS, Jamie Rees, jedwards, Jeff Langemeier, JGreenwell, JHS, jjwatt, JKillian, JNat, joel3000, John Slegers, Jon, jonrsharpe, Josh Caswell, JRodDynamite, Julian, justhalf, Kamyar Ghasemlou, kdopen, Kevin Brown, KIDJourney, Kwarrtz, Lafexlos, lapis, Lee Netherton, Liteye, Locane, Lyndsy Simon, machine yearning, Mahdi, Marc, Markus Meskanen, Martijn Pieters, Matt, Matt Giltaji, Matt S,

		Mattew Whitt, Maximillian Laumeister, mbrig, Mirec Miskuf, Mitch Talmadge, Morgan Thrapp, MSeifert, muddyfish, n8henrie, Nathan Arthur, nehemiah, noɥʎdʎzɐɹƆ, Or East, Ortomala Lokni, pabouk, Panda, Pasha, pktangyue, Preston, Pro Q, pylang, R Nar, Rahul Nair, rap-2-h, Riccardo Petraglia, rll, Rob Fagen, rrao, Ryan Hilbert, Ryan Smith, ryanyuyu, Samuel McKay, sarvajeetsuman, Sayakiss, Sebastian Kreft, Shoe, SHOWMEWHATYOUGOT, Simplans, Slayther, Slickytail, solidcell, StuxCrystal, sudo bangbang, Sunny Patel, SuperBiasedMan, syb0rg, Symmitchry, The_Curry_Man, theheadofabroom, Thomas Gerot, Tim McNamara, Tom Barron, user2314737, user2357112, Utsav T, Valentin Lorentz, Veedrac, viveksyngh, vog, W.P. McNeill, Will, Will, Wladimir Palant, Wolf, XCoder Real, yurib, Yury Fedorov, Zags, Zaz
115	Listas enlazadas	Nemo
116	Llama a Python desde C #	Julij Jegorov
117	Maldiciones básicas con pitón	4444, Guy, kollery, Vinzee
118	Manipulando XML	4444, Brad Larson, Chinmay Hegde, Francisco Guimaraes, greuze, heyhey2k, Rob Murray
119	Matemáticas complejas	Adeel Ansari, Bosoneando, bpachev
120	Matraz	Stephen Leppik, Thomas Gerot
121	Matrices multidimensionales	boboquack, Buzz, rrao
122	Metaclases	2Cubed, Amir Rachum, Antoine Pinsard, Camsbury, Community, driax, Igor Raush, InitializeSahib, Marco Pashkov, Martijn Pieters, Mattew Whitt, OozeMeister, Pasha, Paulo Scardine, RamenChef, Rob Bednark, Simplans, sisanared, zvone
123	Método Anulando	DeepSpace, James
124	Métodos de cuerda	Amitay Stern, Andy Hayden, Ares, Bhargav Rao, Brien, BusyAnt, Cache Staheli, caped114, ChaoticTwist, Charles, Dartmouth, David Heyman, depperm, Doug Henderson, Elazar , ganesh gadila, ghostarbeiter, GoatsWearHats, idjaw, Igor Raush, Ilia Barahovski, j__, Jim Fasarakis Hilliard, JL Peyret, Kevin Brown, krato, MarkyPython, Metasomatism, Mikail Land, MSeifert, mu , Nathaniel Ford, OliPro007, orvi, pzp, ronrest,

		Shrey Gupta, Simplans, SuperBiasedMan, theheadofabroom, user1349663, user2314737, Veedrac, WeizhongTu, wnnmaw
125	Métodos definidos por el usuario	Alessandro Trinca Tornidor, Beall619, mnoronha, RamenChef, Stephen Leppik, Sun Qingyao
126	Mixins	Doc, Rahul Nair, SashaZd
127	Modismos	Benjamin Hodgson, Elazar, Faiz Halde, J F, Lee Netherton, loading..., Mister Mister
128	Módulo aleatorio	Alex Gaynor, Andrzej Pronobis, Anthony Pham, Community, David Robinson, Delgan, giucal, Jim Fasarakis Hilliard, michaelrbock, MSeifert, Nobilis, ppperry, RamenChef, Simplans, SuperBiasedMan
129	Módulo asyncio	2Cubed, Alessandro Trinca Tornidor, Cimbali, hiro protagonist, obust, pylang, RamenChef, Seth M. Larson, Simplans, Stephen Leppik, Udi
130	Módulo de cola	Prem Narain
131	Módulo de colecciones	asmeurer, Community, Elazar, jmunsch, kon psych, Marco Pashkov, MSeifert, RamenChef, Shawn Mehan, Simplans, Steven Maude, Symmitchry, void, XCoder Real
132	Módulo de funciones	Alessandro Trinca Tornidor, enrico.bacis, flamenco, RamenChef, Shrey Gupta, Simplans, Stephen Leppik, StuxCrystal
133	Módulo de matemáticas	Anthony Pham, ArtOfCode, asmeurer, Christofer Ohlsson, Ellis , fredley, ghostarbeiter, Igor Raush, intboolstring, J F, James Elderfield, JGreenwell, MSeifert, niyasc, RahulHP, rajah9, Simplans, StardustGogeta, SuperBiasedMan, yurib
134	Módulo de navegador web	Thomas Gerot
135	Módulo Deque	Anthony Pham, BusyAnt, matsjoyce, ravigadila, Simplans, Thomas Ahle, user2314737
136	Módulo Itertools	ADITYA, Alessandro Trinca Tornidor, Andy Hayden, balki, bpachev, Ffisegydd, jackskis, Julien Spronck, Kevin Brown, machine yearning, nlsdfnbch, pylang, RahulHP, RamenChef, Simplans, Stephen Leppik, Symmitchry, Wickramaranga, wnnmaw
137	Módulo JSON	Indradhanush Gupta, Leo, Martijn Pieters, pzp, theheadofabroom, Underyx, Wolfgang

138	Módulo operador	MSeifert
139	módulo pyautogui	Damien, Rednivrug
140	Módulo Sqlite3	Chinmay Hegde, Simplans
141	Multihilo	Alu, cldseed, juggernaut, Kevin Brown, Kristof, mattgathu, Nabeel Ahmed, nlsdfnbch, Rahul, Rahul Nair, Riccardo Petraglia, Thomas Gerot, Will, Yogendra Sharma
142	Multiprocesamiento	Alon Alexander, Nander Speerstra, unutbu, Vinzee, Will
143	Mutable vs Inmutable (y Hashable) en Python	Cilyan
144	Neo4j y Cypher usando Py2Neo	Wingston Sharon
145	Nodo de lista enlazada	orvi
146	Objetos de propiedad	Alessandro Trinca Tornidor, Darth Shadow, DhiaTN, J F, Jacques de Hooge, Leo, Martijn Pieters, mnoronha, Priya, RamenChef, Stephen Leppik
147	Operadores booleanos	boboquack, Brett Cannon, Dair, Ffisegydd, John Zwinck, Severiano Jaramillo Quintanar, Steven Maude
148	Operadores de Bitwise	Abhishek Jain, boboquack, Charles, Gal Dreiman, intboolstring , JakeD, JNat, Kevin Brown, Matías Brignone, nemesisfixx, poke, R Colmenares, Shawn Mehan, Simplans, Thomas Gerot, tmr232, Tony Suffolk 66, viveksyngh
149	Operadores matemáticos simples	amin, blueenvelope, Bryce Frank, Camsbury, David, DeepSpace, Elazar, J F, James, JGreenwell, Jon Ericson, Kevin Brown, Lafexlos, matsjoyce, Mechanic, Milo P, MSeifert, numbermaniac, sarvajeetsuman, Simplans, techydesigner, Tony Suffolk 66, Undo, user2314737, wythagoras, Zenadix
150	Optimización del rendimiento	A. Ciclet, RamenChef, user2314737
151	os.path	Claudiu, Fábio Perez, girish946, Jmills, Szabolcs Dombi, VJ Magar
152	Pandas Transform: Preforma operaciones en grupos y concatena	Dee

	los resultados.	
153	Patrones de diseño	Charul, denvaar, djaszczurowski
154	Perfilado	J F, keiv.fly, SashaZd
155	Persistencia Python	RamenChef, user2728397
156	pip: PyPI Package Manager	Andy, Arpit Solanki, Community, InitializeSahib, JNat, Mahdi, Majid, Matt Giltaji, Nathaniel Ford, Rápli András, SerialDev, Simplans, Steve Barnes, StuxCrystal, tlo
157	Plantillas en python	4444, Alessandro Trinca Tornidor, Fred Barclay, RamenChef, Ricardo, Stephen Leppik
158	Polimorfismo	Benedict Bunting, DeepSpace, depperm, Simplans, skrrgwasme, Vinzee
159	PostgreSQL	Alessandro Trinca Tornidor, RamenChef, Stephen Leppik, user2027202827
160	Precedencia del operador	HoverHell, JGreenwell, MathSquared, SashaZd, Shreyash S Sarnayak
161	Procesos e hilos	Claudiu, Thomas Gerot
162	Programación Funcional en Python	Imran Bughio, mvis89, Rednivrug
163	Programación IoT con Python y Raspberry PI	dhimanta
164	py.test	Andy, Claudiu, Ffisegydd, Kinifwyne, Matt Giltaji
165	pyaudio	Biswa_9937
166	pygame	Anthony Pham, Aryaman Arora, Pavan Nath
167	Pyglet	Comrade SparklePony, Stephen Leppik
168	PyInstaller - Distribuir código de Python	ChaoticTwist, Eric, mnoronha
169	Python Lex-Yacc	cldseed
170	Python Requests Post	Ken Y-N, RandomHash
171	Python y Excel	bee-sting, Chinmay Hegde, GiantsLoveDeathMetal, hackvan, Majid, talhasch, user2314737, Will

172	Recolección de basura	bogdanciobanu, Claudiu, Conrad.Dean, Elazar, FazeL, J F, James Elderfield, lukess, muddyfish, Sam Whited, SiggyF, Stephen Leppik, SuperBiasedMan, Xavier Combelle
173	Reconocimiento óptico de caracteres	rassar
174	Recursion	Bastian, japborst, JGreenwell, Jossie Calderon, mbomb007, SashaZd, Tyler Crompton
175	Redes Python	atayenel, ChaoticTwist, David, Geeklhem, mattgathu, mnoronha, thsecmaniac
176	Reducir	APerson, Igor Raush, Martijn Pieters, MSeifert
177	Representaciones de cadena de instancias de clase: métodos __str__ y __repr__	Alessandro Trinca Tornidor, jedwards, JelmerS, RamenChef, Stephen Leppik
178	Sangría	Alessandro Trinca Tornidor, depperm, J F, JGreenwell, Matt Giltaji, Pasha, RamenChef, Stephen Leppik
179	Seguridad y criptografía	adeora, ArtOfCode, BSL-5, Kevin Brown, matsjoyce, SuperBiasedMan, Thomas Gerot, Wladimir Palant, wrwrwr
180	Serialización de datos	Devesh Saini, Infinity, rfkortekaas
181	Serialización de datos de salmuera	J F, Majid, Or East, RahulHP, rfkortekaas, zvone
182	Servidor HTTP de Python	Arpit Solanki, J F, jmunsch, Justin Chadwell, Mark, MervS, orvi , quantummind, Raghav, RamenChef, Sachin Kalkur, Simplans , techydesigner
183	setup.py	Adam Brenecki, amblina, JNat, ravigadila, strpeter, user2027202827, Y0da
184	Similitudes en la sintaxis, diferencias en el significado: Python vs. JavaScript	user2683246
185	Sobrecarga	Andy Hayden, Darth Shadow, ericmarkmartin, Ffisegydd, Igor Raush, Jonas S, jonrsharpe, L3viathan, Majid, RamenChef, Simplans, Valentin Lorentz
186	Sockets y cifrado / descifrado de	Mohammad Julfikar

	mensajes entre el cliente y el servidor	
187	Subcomandos CLI con salida de ayuda precisa	Alessandro Trinca Tornidor, anatoly techtonik, Darth Shadow
188	sys	blubberdiblub
189	tempfile NamedTemporaryFile	Alessandro Trinca Tornidor, amblina, Kevin Brown, Stephen Leppik
190	Tipo de sugerencias	alecxe, Annonymous, Antti Haapala, Elazar, Jim Fasarakis Hilliard, Jonatan, RamenChef, Seth M. Larson, Simplans, Stephen Leppik
191	Tipos de datos de Python	Gavin, lorenzofeliz, Pike D., Rednivrug
192	Tipos de datos inmutables (int, float, str, tuple y frozensets)	Alessandro Trinca Tornidor, FazeL, Ganesh K, RamenChef, Stephen Leppik
193	tkinter	Dartmouth, rlee827, Thomas Gerot, TidB
194	Trabajando alrededor del bloqueo global de intérpretes (GIL)	Scott Mermelstein
195	Trabajando con archivos ZIP	Chinmay Hegde, ghostarbeiter, Jeffrey Lin, SuperBiasedMan
196	Trazado con matplotlib	Arun, user2314737
197	Tupla	Anthony Pham, Antoine Bolvy, BusyAnt, Community, Elazar, James, Jim Fasarakis Hilliard, Joab Mendes, Majid, Md.Sifatul Islam, Mechanic, mezzode, nlsdfnbch, noɥʎdʎzɐɹƆ, Selcuk, Simplans, textshell, tobias_k, Tony Suffolk 66, user2314737
198	Unicode	wim
199	Unicode y bytes	Claudiu, KeyWeeUsr
200	urllib	Amitay Stern, ravigadila, sth, Will
201	Usando bucles dentro de funciones	naren

202	Uso del módulo "pip": PyPI Package Manager	Zydnar
203	Velocidad de Python del programa.	ADITYA, Antonio, Elodin, Neil A., Vinzee
204	Visualización de datos con Python	Aquib Javed Khan, Arun, ChaoticTwist, cledoux, Ffisegydd, ifma
205	Web raspado con Python	alecxe, Amitay Stern, jmunsch, mrtuovinen, Ni., RamenChef, Saiful Azad, Saqib Shamsi, Simplans, Steven Maude, sth, sytech, talhasch, Thomas Gerot
206	Websockets	2Cubed, Stephen Leppik, Tyler Gubala