Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

miércoles, 16 de noviembre de 2016

Iteradores y generadores

Iteradores

En Python existen diferentes estructuras de datos que pueden ser recorridas secuencialmente mediante el uso de bucles. Estos objetos llamados **iteradores**, básicamente, son secuencias, contenedores y ficheros de texto.

La declaración **for/in** se utiliza con frecuencia para recorrer los elementos de distintos tipos de **iteradores**: los caracteres de una cadena, los elementos de una lista o una tupla, las claves y/o valores de un diccionario e incluso las líneas de un archivo:

```
# Recorrer los caracteres de una cadena:

cadena = "Python"
for caracter in cadena:
    print(caracter)

# Recorrer caracteres de cadena anterior, en sentido inverso.

for caracter in cadena[::-1]:
    print(caracter)

# Recorrer los elementos de una lista

lista = ['una', 'lista', 'es', 'un', 'iterable']
for palabra in lista:
    print(palabra)

# Recorrer los elementos de la lista anterior, al revés

for palabra in lista[::-1]:
    print(palabra)

# Obtener índice para recorrer todos los elementos de la lista

for indice in range(len(lista)):
    print (indice, lista[indice])

# Recorrer las claves de un diccionario

artistas = { 'Lorca' : 'Escritor', 'Goya' : 'Pintor'}
for clave, valor in artistas.items():
    print(clave,':',valor)

# Leer las líneas de un archivo de texto, una a una
```

Buscar

Buscar

Python para impacientes

Python
IPython
EasyGUI
Tkinter
JupyterLab
Numpy

Anexos

Guía urgente de MySQL
Guía rápida de SQLite3

Entradas + populares

Dar color a las salidas en la consola
En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

Instalación de Python, paso a paso
Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6 . El propósito de esta entrada es mostrar, pas...

Añadir, consultar, modificar y suprimir elementos en Numpy
Acceder a los elementos de un array. [], [,], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

Variables de control en Tkinter
Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

Cálculo con arrays Numpy
Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebid...

Tkinter: interfaces gráficas en Python
Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario (GUI) pero Tkinter es fácil d...

Operaciones con fechas y horas. Calendarios
Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

Convertir, copiar, ordenar, unir y dividir arrays Numpy
Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

Tkinter: Tipos de ventanas
Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

El módulo random
El módulo random de la librería estándar de Python incluye un conjunto de funciones

```
for linea in open("datos.txt"):
    print(linea.rstrip())
```

La función iter()

La función **iter()** se suele emplear para mostrar cómo funciona en realidad un bucle implementado con **for/in**. Antes del inicio del bucle la función **iter()** retorna el objeto iterable con el método subyacente **__iter__()**. Una vez iniciado el bucle, el método **__next__()** permite avanzar, en cada ciclo, al siguiente elemento hasta alcanzar el último. Cuando el puntero se encuentra en el último elemento si se ejecuta nuevamente el método **__next__()** el programa produce la excepción **StopIteration**:

```
lista = [10, 100, 1000, 10000]
iterador = iter(lista)
try:
    while True:
        print(iterador.__next__())

except StopIteration:
    print("Se ha alcanzado el final de la lista")
```

Implementando una clase para iterar cadenas

Los métodos **__next__()** y **__iter__()** permiten declarar clases para crear iteradores a medida.

```
# Declara clase para recorrer caracteres de cadena de texto
# desde el último al primer carácter

class Invertir:
     def __init__(self, cadena):
         self.cadena = cadena
         self.puntero = len(cadena)
     def __iter__(self):
         return(self)
     def __next__(self):
         if self.puntero == 0:
             raise(StopIteration)
         self.puntero = self.puntero - 1
         return(self.cadena[self.puntero])

# Declara iterable y recorre caracteres

cadena_invertida = Invertir('Iterable')
iter(cadena_invertida)

for caracter in cadena_invertida:
     print(caracter, end=' ')

# Devuelven caracteres que restan por iterar (ninguno):

print(list(cadena_invertida.__iter__()))  # []
```

La función range()

Cuando se desea ejecutar un bucle un número de veces determinado se suele utilizar la función **range()** que genera un rango de valores numéricos iterables que no necesitan ser almacenados en una lista o tupla.

```
for elemento in range(1, 11):
    print(elemento, end=' ')  # 1 2 3 4 5 6 7 8 9 10

for elemento in range(10, 0, -1):
    print(elemento, end=' ')  # 10 9 8 7 6 5 4 3 2 1
```

Generadores

Los **generadores** son una forma sencilla y potente de iterador. Un generador es una función especial que produce secuencias completas de resultados en lugar de ofrecer un único valor. En apariencia es como una función típica pero en lugar de devolver los valores con **return** lo hace con la declaración **yield**. Hay que precisar que el término generador define tanto a la propia función como al resultado que produce.

que permiten obtener de distintos modos números a...

Archivo

noviembre 2016 (1)

python.org

pypi.org

Sitios

- ActivePython
- Anaconda
- Bpython
- Django
- Flask
- Ipython
- IronPython
- Matplotlib
- MicroPython
- Numpy
- Pandas
- Pillow
- PortablePython
- PyBrain
- PyCharm
- PyDev
- PyGame
- Pypi
- PyPy
- Pyramid
- Python.org
- PyTorch
- SciPy.org
- Spyder
- Tensorflow
- TurboGears

Una característica importante de los generadores es que tanto las variables locales como el punto de inicio de la ejecución se guardan automáticamente entre las llamadas sucesivas que se hagan al generador, es decir, a diferencia de una función común, una nueva llamada a un generador no inicia la ejecución al principio de la función, sino que la reanuda inmediatamente después del punto donde se encuentre la última declaración **yield** (que es donde terminó la función en la última llamada).

```
# Declara generador

def gen_basico():
    yield "uno"
    yield "dos"
    yield "tres"

for valor in gen_basico():
    print(valor)  # uno, dos, tres

# Crea objeto generador y muestra tipo de objeto

generador = gen_basico()
print(generador)  # generator object gen_basico at 0x7f75ffad55e8
print(type(generador))  # class 'generator'

# Convierte a lista el objeto generador y muestra elementos

lista = list(generador)
print(lista)  # ['uno', 'dos', 'tres']
print(type(lista))  # class 'list'
```

El siguiente generador produce una sucesión de 10 valores numéricos a partir de un valor inicial. El valor final se obtiene sumando 10 al inicial y el bucle se ejecuta mientras el valor inicial es menor que el final. El ejemplo muestra como se almacenan los valores de las variables en cada ciclo y el punto donde se reanuda el bucle en cada llamada.

```
def gen_diez_numeros(inicio):
    fin = inicio + 10
    while inicio < fin:
        inicio+=1
        yield inicio, fin

for inicio, fin in gen_diez_numeros(23):
    print(inicio, fin)
```

En un generador la declaración **yield** puede aparecer en varías líneas e incluso dentro de un bucle. El intérprete Python producirá una excepción de tipo **StopIteration** si encuentra el comando **return** durante la ejecución de un generador.

Relacionado:

- Bucles eficientes con itertools
- Programación funcional. Funciones de orden superior
- Cadenas, listas, tuplas, diccionarios y conjuntos (set)

Ir al índice del tutorial de Python

Publicado por Pherkad en 11:17

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua