Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

domingo, 2 de febrero de 2014

Programación funcional. Funciones de orden superior

Python tiene algunas características de la programación funcional, un paradigma de programación basado en un concepto de función más cercano al matemático que al que habitualmente estamos habituados en la programación imperativa.

Python proporciona algunas funciones de orden superior y permite también definir otras a medida. El siguiente ejemplo construye un conversor de números anidando varias funciones en una **función de orden superior**.

```
def conversor(sis):
    def sis_bin(numero):
        print('dec:', numero, 'bin:', bin(numero))

    def sis_oct(numero):
        print('dec:', numero, 'oct:', oct(numero))

    def sis_hex(numero):
        print('dec:', numero, 'hex:', hex(numero))

    sis_func = {'bin': sis_bin, 'oct': sis_oct, 'hex': sis_hex}
    return sis_func[sis]

# Crea una instancia del conversor hexadecimal
conversorhex = conversor('hex')

# Convierte 100 de decimal a hexadecimal
conversorhex(100)

# Otro modo de usar el conversor.
# Convierte 9 de decimal a binario
conversor('bin')(9)
```

La función map()

La función de orden superior **map**() aplica una función a una lista de datos y devuelve un iterador que contiene todos los resultados para los elementos de la lista.

En el siguiente ejemplo la función **cuadrado** calcula el cuadrado de un número. La **lista1** contiene una lista de datos numéricos. Con **map**(cuadrado, lista1) se aplica la función cuadrado a cada elemento de la lista.

```
def cuadrado(numero):
    return numero ** 2

lista1 = [-2, 4, -6, 8]

# Convierte a lista el iterador obtenido
lista2 = list(map(cuadrado, lista1))

# Muestra elementos de la lista
print(lista2)  # 4, 16, 36, 64
```

Para calcular el área de un circulo necesitamos el número pi que esta disponible, con un número suficiente de decimales, en el módulo de la biblioteca estándar math. En el siguiente ejemplo la función **area_circulo** calcula el área de un circulo. La **lista3** tiene una lista de datos numéricos que son longitudes de radios diferentes. Con **map**(area_circulo, lista3) se aplica la función a cada elemento de la lista.

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

```
import math

def area_circulo(radio):
    return math.pi * radio ** 2

lista3 = [1, 2, 3]

# Devuelve iterador que es convertido a lista
lista4 = list(map(area_circulo, lista3))
print(lista4)
```

Para convertir el iterador a una lista hemos empleado la función **list**(). También, podríamos haber recorrido el iterador con un **for...in**:

```
for resultado in map(area_circulo, lista3):
    print(resultado)
```

La función filter()

La función **filter**() aplica un filtro a una lista de datos y devuelve un iterador con los elementos que superan el filtro.

```
# La función verifica si un número es negativo
def esneg(numero):
    # Devuelve True/False según sea o no nº negativo
    return (numero < 0)

lista5 = [-3, -2, 0, 1, 9, -5]

# Muestra los números negativos de la lista
# La función esneg() es llamada para comprobar,
# uno a uno, todos los números de la lista
print(list(filter(esneg, lista5)))
```

La función reduce()

La función **reduce**() aplica una función a una lista de datos evaluando los elementos por pares. La primera vez se aplica al primer y segundo elemento, la siguiente, se aplicará al valor devuelto por la función junto al tercer elemento y así, sucesivamente, reduciendo la lista hasta que quede un sólo elemento.

A partir de Python 3 si queremos utilizar reduce() debemos importar el módulo functools:

```
import functools

def multiplicar(x, y):
    print(x * y)  # muestra el resultado parcial
    return x * y

lista = [1, 2, 3, 4]
valor = functools.reduce(multiplicar, lista)
print(valor)  # muestra el resultado final
```

La función lambda

La función **lambda** se utiliza para declarar funciones que sólo ocupan una línea. Son objetos que se pueden asignar a variables para usar con posterioridad.

```
cuadrado = lambda x: x*x

lista = [1,2,3,5,8,13]
for elemento in lista:
    print(cuadrado(elemento))

# Lambda, con 2 argumentos:

area_triangulo = (lambda b,h: b*h/2)
medidas = [(34, 8), (26, 8), (44, 18)]
for datos in medidas:
    base = datos[0]
    altura = datos[1]
    print(area_triangulo(base, altura))
```

que permiten obtener de distintos modos números a...

Archivo

febrero 2014 (17)

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

Comprensión de listas

Es un tipo de construcción que consta de una expresión que determina como modificar los elementos de una lista, seguida de una o varias clausulas **for** y, opcionalmente, una o varias clausulas **if**. El resultado que se obtiene es una lista, un diccionario o un conjunto.

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

# Cada elemento de la lista se eleva al cubo
cubos = [valor ** 3 for valor in lista]
print('Cubos de 1 a 10:', cubos)


numeros = [135, 154, 180, 193, 210]
divisiblespor3 = [valor for valor in numeros if valor % 3.0 == 0]

# Muestra lista con los números divisibles por 3
print(divisiblespor3)


# Define función devuelve el inverso de un número
def funcion(x):
    return 1/x

L = [1, 2, 3]  # declara lista

# Muestra lista con inversos de cada número
print([funcion(i) for i in L])

# Devuelve un conjunto (elementos no repetidos)
z = {elem for elem in [1, 2, 1, 2, 3, 1]}
print(z)  # {1, 2, 3}

# Devuelve un diccionario
dicc1 = {num: num**2 for num in range(1, 6)}
print(dicc1)  # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Generadores

Los generadores funcionan de forma parecida a la comprensión de listas pero no devuelven listas sino generadores. Un generador es una clase especial de función que genera valores sobre los que iterar. La sintaxis usada es como la usada en la comprensión de listas pero en vez de usar corchetes se utilizan paréntesis. Para devolver los valores se utiliza **yield** en vez de **return**.

```
# Define generador
def generador(inicio, fin, incremento):
    while(inicio <= fin):
        yield inicio  # devuelve valor
        inicio += incremento

# Recorre los valores del generador
for valor in generador(0, 6, 1):
    # Muestra valores, uno a uno:
    print(valor)  # 0 1 2 3 4 5 6

# Obtiene una lista del generador
lista = list(generador(0, 8, 2))

# Muestra lista
print(lista)  # [0,2,4,6,8]
```

La función Decorador

Es una función que recibe una función como parámetro y devuelve otra función como valor de retorno. Se utiliza cuando es necesario definir varias funciones que son muy parecidas. La función devuelta actúa como un envoltorio (wrapper) resolviendo lo que sería común a todas las funciones. También se aplica a clases.

```
# Define decorador
def decorador1(funcion):
    # Define función decorada
    def funciondecorada(*param1, **param2):
        print('Inicio', funcion.__name__)
        funcion(*param1, **param2)
```

```
        print('Fin', funcion.__name__)
    return funciondecorada

def suma(a, b):
    print(a + b)

suma2 = decorador1(suma)
suma2(1,2)
suma3 = decorador1(suma)
suma3(2,2)

# Otra forma más elegante, usando @:

@decorador1
def producto(arg1, arg2):
    print(arg1 * arg2)

producto(5,5)


# El siguiente decorador genera tablas de sumas
# y multiplicaciones. El código que es común a todas
# las funciones se declara en la función 'envoltura':

def tablas(funcion):
    def envoltura(tabla=1):
        print('Tabla del %i:' %tabla)
        print('-' * 15)
        for numero in range(0, 11):
            funcion(numero, tabla)
        print('-' * 15)
    return envoltura

@tablas
def suma(numero, tabla=1):
    print('%2i + %2i = %3i' %(tabla, numero, tabla+numero))

@tablas
def multiplicar(numero, tabla=1):
    print('%2i X %2i = %3i' %(tabla, numero, tabla*numero))

# Muestra la tabla de sumar del 1
suma()

# Muestra la tabla de sumar del 4
suma(4)

# Muestra la tabla de multiplicar del 1
multiplicar()

# Muestra la tabla de multiplicar del 10
multiplicar(10)
```

Ir al índice del tutorial de Python

Publicado por Pherkad en 4:35

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua