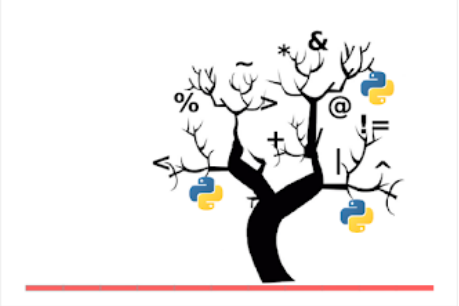
Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

miércoles, 6 de abril de 2016

Operadores estándar como funciones



El módulo **operator** de la librería estándar de Python ofrece un conjunto de funciones eficientes que tienen su correspondencia en los operadores comunes de Python que se utilizan en operaciones de comparación, aritméticas, lógicas y con secuencias. Además, este módulo incluye algunas herramientas para hacer referencia con facilidad a datos organizados en listas y a atributos de clases; y otro grupo de funciones para operaciones *in situ*.

Un ejemplo de correspondencia sería la función **add**() que es equivalente al operador aritmético de la suma "**+**".

Asimismo, la mayoría de las funciones utilizan los mismos nombres de los métodos especiales de las clases; pero sin incluir los dobles guiones bajos "**__**" que preceden y siguen a dichos nombres.

Las funciones del módulo **operator** se agrupan en las siguientes categorías:

Funciones para comparar objetos

Las funciones para operaciones de comparación pueden utilizarse con cualquier tipo de objeto y devuelven como resultado el valor **True** o **False**.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#

import operator as op

x = 5
y = 4
z = 5

# Comparación de números:

print(op.lt(x, y))  # Menor que: False
print(op.le(x, y))  # Menor o igual que: False
print(op.eq(x, z))  # Igual que: True
print(op.ne(x, z))  # Distinto que: False
print(op.ge(x, z))  # Mayor o igual que: True
print(op.gt(z, y))  # Mayor que: True
print()

q = 'A'
r = 'B'
s = 'A'

# Comparación de cadenas:

print(op.lt(q, r))  # Menor que: True
print(op.le(q, r))  # Menor o igual que: True
```

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...

Threading: programación con hilos (I)
En programación, la técnica que permite que una aplicación ejecute

```
print(op.eq(q, s))  # Igual True
print(op.ne(q, s))  # Distinto que False
print(op.ge(q, s))  # Mayor o igual que: True
print(op.gt(s, r))  # Mayor que: False
print()

# Comparación utilizando métodos de clases equivalentes:

print(op.__lt__(x, y))  # Menor que: False
print(op.__le__(x, y))  # Menor o igual que: False
print(op.__eq__(x, z))  # Igual que: True
print(op.__ne__(x, z))  # Distinto que: False
print(op.__ge__(x, z))  # Mayor o igual que: True
print(op.__gt__(z, y))  # Mayor que: True
```

Funciones para operaciones lógicas

Las funciones para operaciones lógicas también son aplicables en general a todos los objetos y soportan pruebas de verdad, de identidad y operaciones booleanas.

```
# Función truth(): Se utiliza para realizar pruebas de verdad
# La función devuelve False si el objeto tiene alguno de
# los siguientes valores: None, False, 0, '' y vacío.

class MiClase:
 pass

obj1 = MiClase()
obj2 = None
obj3 = MiClase()
list1 = []
list2 = [1, 2]
num1 = 0
num2 = 523

print(op.truth(obj1))  # True
print(op.truth(obj2))  # False
print(op.truth(obj3))  # True
print(op.truth(list1))  # False
print(op.truth(list2))  # True
print(op.truth(num1))  # False
print(op.truth(num2))  # True
print()

# Función not_(): Devuelve el valor opuesto (negado) al obtenido

log1 = False
num1 = 0
obj1 = MiClase()
tup1 = ()
tup2 = (1, 2, 3)
tup3 = ()

print(op.not_(log1))  # True
print(op.not_(num1))  # True
print(op.not_(obj1))  # False
print(op.not_(tup1))  # True
print(op.not_(tup2))  # False
print(op.not_(tup3))  # True
print()

# Función is_(a, b) "is": La función compara dos objetos y
# devuelve True si el objeto 'a' se corresponde con el 'b',
# o bien, si tienen el mismo valor. En caso contrario
# la función devuelve False.

obj2 = obj1
cad1 = "py"
cad2 = "pyc"

print(op.is_(obj1, obj2))  # True
print(op.is_(obj1, obj3))  # False
print(op.is_(tup1, tup3))  # True
print(op.is_(cad1, cad2))  # False
print()

# Función is_not(a, b) "is not": La función compara dos objetos y
# devuelve False si el objeto 'a' se corresponde con el 'b',
# o bien, si tienen el mismo valor. En caso contrario
```

simultáneamente varias operaciones en el mismo espacio de proceso se...

Archivo

abril 2016 (2)

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

```
# la función devuelve True. Es la función opuesta a is_()

print(op.is_not(obj1, obj2))  # False
print(op.is_not(obj1, obj3))  # True
print(op.is_not(tup1, tup3))  # False
print(op.is_not(cad1, cad2))  # True
```

Funciones para operaciones matemáticas

El mayor número de funciones se agrupan en este apartado y, básicamente, se dedican a operaciones aritméticas y binarias.

```
a = 5
b = 3
c = 0o52     # octal
d = 0b101010 # bin(42)

obj1 = -4.5

import numpy as np
m1 = np.array([[1, 2], [3, 4]])
m2 = np.array([[2, 2], [2, 2]])

# Suma '+': op.add(a, b) y op.__add__(a, b)
print(op.add(a, b))  # 8
# Resta '-': op.sub(a, b) y op.__sub__(a, b)
print(op.sub(a, b))  # 2
# Multiplicación '*':  op.mul(a, b) y op.__mul__(a, b)
print(op.mul(a, b))  # 15
# División con decimales '/': op.truediv(a, b) y op.__truediv__(a, b)
print(op.truediv(a, b))  # 1.6666666666666667
# Potenciación '**': op.pow(a, b) y op.__pow__(a, b)
print(op.pow(a, b))  # 125
# Resto de división '%': op.mod(a, b) y op.__mod__(a, b)
print(op.mod(a, b))  # 2
# Cociente entero '//': op.floordiv(a, b) y op.__floordiv__(a, b)
print(op.floordiv(a, b))  # 1
# Multiplicación de matrices "@" (Python 3.5+):
op.matmul(m1, m2) y op.__matmul__(m1, m2)
print(op.matmul(m1, m2))  # [[6, 6], [14, 14]]
# Valor absoluto: op.abs(obj) y op.__abs__(obj)
print(op.abs(obj1))  # 4.5
# Valor positivo: op.pos(obj) y op.__pos__(obj)
print(op.pos(obj1))  # -4.5
# Valor negativo: op.neg(obj) y op.__neg__(obj)
print(op.neg(obj1))  # 4.5
# Convierte valor a entero: op.index(c) y op.__index__(a)
print(op.index(c))  # 42 en dec
# Operación AND '&': op.and_(a, b) y op.__and__(a, b)
print(op.and_(a, b))  # 101 and 011 = 001 - 1
# Operación OR '|': op.or_(a, b) y op.__or__(a, b)
print(op.or_(a, b))  # 101 or 011 = 111 - 7
# Operación XOR: op.xor(a, b) y op.__xor__(a, b)
print(op.xor(a, b))  # 101 xor 011 = 110 - 6
# Desplazamiento a la derecha: op.rshift(a, b) y op.__rshift__(a, b)
print(op.rshift(d, 1))  # 0b10101 - 21
# Desplazamiento a la izquierda: op.lshift(a, b) y op.__lshift__(a, b)
print(op.lshift(d, 1))  # 0b1010100 - 84
# Invertir '~': op.inv(obj), op.invert(obj),
# op.__inv__(obj) y op.__invert__(obj)
print(op.inv(a), bin(op.inv(a)))  # 0b101 - 0b110 = -6 -0b110
```

Funciones para operaciones con secuencias

Estas funciones son para realizar operaciones con cadenas, listas y tuplas.

```
a = "abcabcabc"
b = "a"
x = None
list1 = ['a', 'b', 'c', 'd']

# Concatenar '+': op.concat(a, b) y op.__concat__(a, b)
print(op.concat(a, b))  # 'abcabcabc' + 'a' = 'abcabcabca'
# Comprobar si 'b' existe en 'a':  op.contains(a, b) y
# op.__contains__(a, b)
print(op.contains(a, b))  # True
```

```
# Contar número de apariciones de 'b' en 'a': op.countOf(a, b)
print(op.countOf(a, b))  # 3
# Borrar elemento por posición: op.delitem(a, b) y
# op.__delitem__(a, b)
op.delitem(list1, 2)     # borra el tercer elemento de la lista
print(list1)  # muestra elementos que quedan en la lista:
# ['a', 'b', 'd']
# Obtener elemento por su posición: op.getitem(a, b) y
# op.__getitem__(a, b)
print(op.getitem(list1, 2))  # obtiene 3er elemento: 'd'
# Obtener índice del primer elemento encontrado: op.indexOf(a, b)
print(op.indexOf(list1, b))  # 0
# Establecer elemento de la posición indicada con nuevo valor:
# op.setitem(a, b, c) y op.__setitem__(a, b, c)
op.setitem(list1, 2, 'e')  # Asigna al 3er elemento 'e'
# Muestra lista después del cambio anterior:
print(list1)  # ['a', 'b', 'e']
# Devolver longitud estimada: op.length_hint(obj, default=0)
# Si no puede obtener ningún valor, devolverá el predeterminado
print(op.length_hint(list1))  # 3
print(op.length_hint(x, 10))  # 10
```

Funciones para hacer referencia a atributos y elementos

El módulo **operator** incluye funciones para hacer referencia a atributos de clases y a elementos organizados en listas y tuplas. Son útiles como mecanismo para acceder a campos que pueden emplearse como argumentos de otras funciones como **map**(), **sort**(), **itertools**.**groupby**() y otras.

```
# Funciones op.itemgetter(item) y op.itemgetter(*items):
# Se utilizan para realizar ciertas operaciones en las que
# se emplean referencias a elementos. En el siguiente
# ejemplo se utiliza la referencia a los elementos de
# una lista de tuplas. El valor 0 identifica al primer
# elemento de cada tupla (nombre de horzaliza) y el valor
# 1 al segundo elemento (peso). En este caso estos valores
# sirven para definir la clave (key) para ordenar los
# artículos

# Declara lista de tuplas:
huerta = [('cebollas', 100), ('tomates', 210), ('pimientos', 60)]
print('Ordenar lista por nombre:',
      sorted(huerta, key=op.itemgetter(0)))
# [('cebollas', 100), ('pimientos', 60), ('tomates', 210)]
print('Ordenar lista por peso  :',
      sorted(huerta, key=op.itemgetter(1)))
# [('pimientos', 60), ('cebollas', 100), ('tomates', 210)]
print()

# Funciones op.itemgetter(item) y op.itemgetter(*items):
# Se utilizan para realizar operaciones utilizando los nombres de
# los atributos de una clase.

# Define Clase
class Huerta:
    def __init__(self, hortaliza, peso, precio):
        self.hortaliza = hortaliza
        self.peso =  peso
        self.precio = precio
    def __repr__(self):
        return repr((self.hortaliza, self.peso, self.precio))
    def calcular(self):
        return self.peso * self.precio

# Añadir a lista varios objetos de la clase Huerta:
hortalizas = [Huerta('cebollas', 100, 0.80),
              Huerta('tomates', 210, 0.90),
              Huerta('pimientos', 60, 0.45)]
# Ordenar objetos por dos de sus atributos: 'peso' y 'precio'
print('Ordenar por peso:  ',
      sorted(hortalizas, key=op.attrgetter('peso', 'precio')))
# [('pimientos', 60, 0.45), ('cebollas', 100, 0.8),
# ('tomates', 210, 0.9)]
print()

# Función op.methodcaller(name[, args...])
# Se utiliza para realizar operaciones llamando a métodos y
# funciones. En el siguiente ejemplo utiliza el valor devuelto
# por un método para ordenar una lista de objetos de mayor a
```

```
# menor valor
lista = sorted(hortalizas,
               key=op.methodcaller('calcular'),
               reverse=True)
for elemento in lista:
    print(elemento.hortaliza, elemento.peso * elemento.precio)
print()

# En el ejemplo siguiente se llama a la función replace() para
# buscar y sustituir los espacios de una secuencia (cadena):
cad1 = 'Python para impacientes'
sustituir = op.methodcaller('replace', ' ', '.+_+.')
print(sustituir(cad1))  # Python.-_-.para.-_-.impacientes
```

Funciones de operadores *in situ*

Muchas operaciones tienen una versión de función llamada *in situ* que proporcionan un acceso a los operadores más elemental como alternativa al uso de la sintaxis ordinaria.

Por ejemplo: la declaración de '**a *= b**' es equivalente a '**a = operator.imul(a, b)**'. Otra forma de expresarlo es **c = operator.iadd(a, b)**, que es equivalente a **c = a; c += b**.

En estos casos hay que tener en cuenta que cuando se llama al método, el cálculo y la asignación se hacen en dos etapas diferenciadas. Sin embargo, cuando se utilizan las funciones *in situ* sólo se realiza la primera etapa.

Para objetos inmutables tales como cadenas, números y tuplas el nuevo valor se calcula pero no se asigna a la variable de entrada:

```
blog = 'Python'
iadd(blog, ' para impacientes')  # 'Python para impacientes'
print(blog)  # 'Python'
```

Sin embargo, con listas y diccionarios la función *in situ* realiza directamente la asignación:

```
c = ['a', 'b', 'c']
iadd(c, [' ', 'x', 'y', 'z'])  # ['a', 'b', 'c', ' ', 'x', 'y', 'z']
print(c)  # ['a', 'b', 'c', ' ', 'x', 'y', 'z']
```

Funciones *in situ:*

```
# Funciones de operadores in situ

a = 8
b = 4

# Suma "+=": op.iadd(a, b) y op.__iadd__(a, b)
# a = iadd(a, b) es equivalente a += b

a = op.iadd(a, b)  # 8 + 4
print(a)  # 12

# Resta "-=": op.isub(a, b) y op.__isub__(a, b)
# a = isub(a, b) es equivalente a -= b

a = op.isub(a, b)  # 12 - 4
print(a)  # 8

# Multiplicación "*=": op.imul(a, b) y op.__imul__(a, b)
# a = imul(a, b) es equivalente a *= b

a = op.imul(a, b)  # 8 * 4
print(a)  # 32

# División "/=": op.itruediv(a, b) y op.__itruediv__(a, b)
# a = itruediv(a, b) es equivalente a /= b.

a = op.itruediv(a, b)  # 32 / 4
print(a)  # 8.0 (float)

# Potenciación "**=": op.ipow(a, b) y op.__ipow__(a, b)
# a = ipow(a, b) es equivalente a **= b.

a = op.ipow(a, b)  # 8 ** 4 = 8 * 8 * 8 * 8
```

```
print(a)  # 4096.0

# Resto de división "%=": op.imod(a, b) y op.__imod__(a, b)
# a = imod(a, b) es equivalente a %= b.

resto = op.imod(a, b)  # 4096.0 % 4
print(resto)  # 0

# Cociente "//=": op.ifloordiv(a, b) y op.__ifloordiv__(a, b)
# a = ifloordiv(a, b) es equivalente a //= b.

coc = op.ifloordiv(a, b)  # 4096.0 // 4
print(coc)  # 1024.0

import numpy as np
m1 = np.array([[1, 2], [3, 4]])
m2 = np.array([[2, 2], [2, 2]])

# Multiplicación de matrices "@=": op.imatmul(a, b) y
# op.__imatmul__(a, b)
# a = imatmul(a, b) es equivalente a @= b. (Python 3.5+)

m1 = op.matmul(m1, m2)  # m1 @ m2 = [[1, 2], [3, 4]] @ [[2, 2], [2, 2]]
print(m1)  # [[6, 6], [14, 14]]

x = 1
y = 0

# Operación AND "&=": op.iand(a, b) y op.__iand__(a, b)
# a = iand(a, b) es equivalente a &= b.

z = op.iand(x, y)  # 1 & 0 = 0
print(z)  # 0

# Operación OR "|=": op.ior(a, b) y op.__ior__(a, b)
# a = ior(a, b) es equivalente a |= b.

z = op.ior(x, y)  # 1 | 0 = 1
print(z)  # 1

# Operación XOR "^=": op.ixor(a, b) y op.__ixor__(a, b)
# a = ixor(a, b) es equivalente a ^= b.

z = op.ixor(x, y)  # 1 ^ 0 = 1
print(z)  # 1

# Desplazamiento a la derecha: op.irshift(a, b) y
# op.__irshift__(a, b)
# a = irshift(a, b) es equivalente a desplazamiento a la derecha= b.

x1 = 1
y1 = 2

z = op.irshift(x1, y1)  # 1 desplazamiento a la derecha 2 = 0
print(z)  # 0

# Desplazamiento a la izquierda: op.ilshift(a, b) y
# op.__ilshift__(a, b)
# a = ilshift(a, b) es equivalente a desplazamiento a la izquierda= b.

z = op.ilshift(x1, y1)  # 1 desplazamiento a la izquierda 2 = 4
print(z)  # 4

# Concatenar "+=": op.iconcat(a, b) y op.__iconcat__(a, b)
# a = iconcat(a, b) es equivalente a += b siendo a y b secuencias

a = 'abcabcabc'
b = 'a'

a = op.iconcat(a, b)  # 'abcabcabc' + 'a' -> 'abcabcabca'
print(a)  # 'abcabcabca'
```

Ir al índice del tutorial de Python

Publicado por Pherkad en 15:45

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua