

Programación funcional con Python (II)

Más sobre funciones de orden superior

Las funciones de orden superior pueden recibir funciones como parámetros, y/o devolver funciones.

En la parte 1 vimos algunas incorporadas en Python: `map` y `filter`.

Pero hay más:

`max()` y `min()` que permiten encontrar el mayor o menor en un iterable:

```
>>> max([2, 8, 9, 3])
9
```

```
>>> max("abcba")
'c'
```

```
>>> min([2, 8, 9, 3])
2
>>> min("abcba")
'a'
```

```
>>> articulos = ( (101, "arvejas", 53.0), (220, "choclo", 60.0), (150,
"garbanzos", 75.5))
>>> max(articulos)
(220, 'choclo', 60.0)
>>> min(articulos)
(101, 'arvejas', 53.0)
```

En los ejemplos anteriores, reciben iterables solamente... pero están diseñadas para recibir una función para re-definir cómo se ponderan los elementos.

En el diccionario del ejemplo anterior ¿Necesitamos obtener el artículo con mayor precio? (por defecto `max` devolvió el mayor código)

```
>>> max(articulos, key=lambda x: x[2])
(150, 'garbanzos', 75.5)
```

De esta manera, utilizando una función como parámetro se le especificó a `max` cómo evaluar el "mayor"

Otro ejemplo:

`caminos = [ (1, 4, 5), (7, 1, 1), (1, 8, 2), (1, 3, 4)]` es una lista de tuplas, cada una con tres elementos, correspondientes a distancias recorridas...

Si queremos encontrar la mayor distancia total, podríamos usar

```
maximo = 0
recorrido_maximo = ()
for recorrido in caminos:
    actual = sum(recorrido)
    if actual > maximo:
        maximo = actual
        recorrido_maximo = recorrido

print(recorrido_maximo)
```

O, usando la capacidad de función de orden superior de `max` podemos escribir solamente:

```
print( max(caminos, key=lambda x: sum(x)) )
```

que no solo es más concisa si no que es más fácil de leer.

¿Y si necesitamos una lista de los caminos, pero con sus tramos ordenados de menor a mayor? Algo así:

```
[ (1, 4, 5), (1, 1, 7), (1, 2, 8), (1, 3, 4)]
```

Para ordenar podemos usar `sorted()`

Y en conjunto con comprensión de listas quedaría:

```
[sorted(recorrido) for recorrido in caminos]
```

¿Y esa lista ordenada de menor a mayor según la distancia total? Algo así:

```
[[1, 3, 4], [1, 1, 7], [1, 4, 5], [1, 2, 8]]
```

(porque $1+3+4 < 1+1+7 < 1+4+5 < 1+2+8$)

Para esto es bueno saber que `sorted()` es **también** función de **orden superior**... eso permite especificar el criterio de ordenamiento:

```
# lista de caminos con sus tramos ordenados
aux = [sorted(recorrido) for recorrido in caminos]
```

```
# lista de caminos con tramos ordenados,
# ordenada de menor a mayor distancia total
sorted(aux, key=lambda x: sum(x))

# o todo en una línea
sorted([sorted(recorrido) for recorrido in caminos], key=lambda x:
sum(x))
```

Si no necesitamos reutilizar la función que le pasamos a una función de orden superior, podemos usar lambdas como en los ejemplos de arriba.

Otra opción puede ser definir una función no anónima ("tradicional")

```
def distancia_total(t):
    return sum(t)
```

o también podemos asignar la función anónima a una "variable":

```
distancia_total = lambda x: sum(x)
```

De cualquiera de estas formas, hacemos:

```
sorted([sorted(recorrido) for recorrido in caminos], key=distancia_total)
```

itertools

Es un módulo de la biblioteca estándar de Python que nos provee funciones para usar con iterables.

`import itertools` o similar para utilizar estas funciones.

● Iteradores infinitos:

`count()` es una versión infinita de `range()`

`count(5)` cuenta **desde 5**

`count(5, 3)` cuenta desde 5, de 3 en 3... : 5, 8, 11, 14 ...

`cycle()` repite una secuencia indefinidamente

```

>>> a = itertools.cycle('xyz')
>>> next(a)
'x'
>>> next(a)
'y'
>>> next(a)
'z'
>>> next(a)
'x'
>>> next(a)
'y'
>>> next(a)

```

repeat() repite un valor indefinidamente, o un número determinado de veces:

```

repeat(9)

>>> list(itertools.repeat(9, 5))
[9, 9, 9, 9, 9]

```

● Iteradores finitos:

accumulate() permite realizar una operación (suma por defecto) a un iterable p , produciendo otro iterable q de la forma $q_0 = p_0$, $q_1 = q_0 + p_1$, $q_2 = q_1 + p_2 \dots$

```

>>> a
[7, 5, 3, 9]
>>> tuple(itertools.accumulate(a))
(7, 12, 15, 24)

```

```

0 + 7 → 7
7 + 5 → 12
12 + 3 → 15
15 + 9 → 24

```

Siguiendo con el ejemplo de los caminos....

```

>>> caminos
[(1, 4, 5), (7, 1, 1), (1, 8, 2), (1, 3, 4)]
>>> tuple(itertools.accumulate((sum(x) for x in caminos)))
(10, 19, 30, 38)

```

... una lista con los subtotales de distancias recorridas en cada tramo.

`compress()`

```
>>> list(itertools.compress( [('a', 2), ('b', 3), ('d', 5), ('b', 0)], [1, 0, 1, 0]))
```

```
[('a', 2), ('d', 5)]
```

Devuelve sólo los elementos del primer iterable que se pasa como argumento en las posiciones donde el segundo iterable sea True

Si tengo:

```
palabra = "ballena"    # palabra incógnita
intento = "lep"        # letras que intentó el usuario
```

y quiero una lista de letras de palabra que no estén en intento:

```
list(itertools.compress(palabra, [True if (x not in intento) else False
for x in palabra ] ))
```

```
['b', 'a', 'n', 'a']
```

→ `[True if (x not in intento) else False for x in palabra ]`
genera

```
[True, True, False, False, False, True, True]
  b     a     l     l         e     n     a
```

Esto hace que de palabra se tome la primera, segunda, sexta y séptima letras...

`takewhile()` / `dropwhile()` copia / descarta elementos **mientras que** la expresión sea verdadera

```
>>> list(itertools.takewhile(lambda x: x <= 7, [4, 1, 7, 2, 9, 7, 3, 8]))
```

```
[4, 1, 7, 2]
```

Copia hasta que encuentra un elemento mayor a 7, (`<= 7` es falso) y descarta desde ese elemento hasta el final.

```
>>> list(itertools.dropwhile(lambda x: x < 6, [4, 1, 7, 2, 9, 7, 3, 8]))
```

```
[7, 2, 9, 7, 3, 8]
```

Descarta hasta que encuentra un elemento igual o mayor a 6 (el primer 7 de la lista...) y copia desde ese elemento hasta el final.

filterfalse()

Es como filter pero permite usar la expresión negada; esto es útil para reutilizar la misma expresión que defina si un elemento queda o no, en lugar de escribir dos...

```
>>> a = [3, 5, 7, 6, 4]
>>> impar = lambda x: x % 2
>>> list(filter(impar, a))
[3, 5, 7]
>>> list(itertools.filterfalse(impar, a))    # los pares
[6, 4]
```

product() nos permite obtener el producto cartesiano entre dos iterables (todas las posibles combinaciones entre elementos, como un join en sql)

```
>>> numeros = [1, 2, 3]
>>> colores = ["rojo", "azul", "verde"]
>>> list(itertools.product(colores, numeros))
[('rojo', 1), ('rojo', 2), ('rojo', 3), ('azul', 1), ('azul', 2),
 ('azul', 3), ('verde', 1), ('verde', 2), ('verde', 3)]
```

Un fixture:

```
jugadores = ["river", "boca", "unión"]
>>> list(itertools.product(jugadores, jugadores))
[('river', 'river'), ('river', 'boca'), ('river', 'unión'), ('boca',
 'river'), ('boca', 'boca'), ('boca', 'unión'), ('unión', 'river'),
 ('unión', 'boca'), ('unión', 'unión')]
```

y eliminando los que tengan dos equipos iguales:

```
>>> list(filter(lambda x: x[0] != x[1], itertools.product(jugadores,
jugadores)))
[('river', 'boca'), ('river', 'unión'), ('boca', 'river'), ('boca',
 'unión'), ('unión', 'river'), ('unión', 'boca')]
```

permutations() - genera todas las permutaciones (n!) posibles con los elementos de un iterable...

Para obtener la lista de posibles formas de empaquetar cuatro productos, sabiendo que un producto A nunca va en los extremos:

```
import itertools

prods = ["A", "B", "C", "D"]

def regla(x):
    return (x[0] != "A" and x[-1] != "A")

posibles = list(filter(regla, itertools.permutations(prods)))

print(posibles)

[('B', 'A', 'C', 'D'), ('B', 'A', 'D', 'C'), ('B', 'C', 'A', 'D'), ('B', 'C', 'D', 'A'), ('B', 'D', 'A', 'C'), ('B', 'D', 'C', 'A'), ('C', 'A', 'B', 'D'), ('C', 'A', 'D', 'B'), ('C', 'B', 'A', 'D'), ('C', 'B', 'D', 'A'), ('C', 'D', 'A', 'B'), ('C', 'D', 'B', 'A'), ('D', 'A', 'B', 'C'), ('D', 'A', 'C', 'B'), ('D', 'B', 'A', 'C'), ('D', 'B', 'C', 'A'), ('D', 'C', 'A', 'B'), ('D', 'C', 'B', 'A')]
```

combinations() genera las combinaciones posibles de los elementos de un iterable en n lugares.

```
>>> list(itertools.combinations(range(1, 5), 3))
[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
```

Con números de 1 al 4 podemos armar 4 combinaciones de 3 números.

Las 2.598.960 combinaciones de 5 cartas de poker:

```
hands = list(combinations(tuple(product(range(13), '♠♥♦♣')), 5))
```

Hay **más funciones en itertools**; se pueden consultar en la documentación oficial

<https://docs.python.org/3/library/itertools.html>

Además, hay una excelente selección "**itertools recipes**" con ejemplos para consultar y estudiar: <https://docs.python.org/3/library/itertools.html#itertools-recipes>

functools

Es un módulo también de la biblioteca estándar que provee operaciones para funciones de orden superior.

import functools o similar para utilizar estas funciones.

partial() - devuelve una función que resulta de la función que se le pasa como parámetro, con alguno(s) argumentos(s) fijos. Permite reutilizar una función con valores preasignados a algunos de sus parámetros.

```
>>> print_list = functools.partial(print, sep="-")
>>> print_list(3, 6, "toto")
3-6-toto
```

En este ejemplo, `print_list` es la función `print` considerando `sep` siempre como `"-"`

Si en un programa de ingreso de pacientes de un hospital tenemos:

```
def agregarPersona(nombre, apellido, edad, dni, hab, cama)
```

podemos aprovecharla para hacer un programa dedicado a neonatología en caso de necesidad, sin necesidad de rehacer la función original...

```
agregarBebe = functools.partial(agregaPersona, edad=0, dni=0)
```

y usamos

```
agregarBebe("Ana", "López", "A123", "2")
```

en lugar de

```
agregarPersona("Ana", "López", 0, "", "A123", "2")
```

reduce() - Aplica una función a dos elementos del iterable, de izquierda a derecha, hasta obtener un único valor.

En versiones anteriores de Python se podía utilizar, al igual que `map` y `filter`, sin necesidad de ningún `import`.

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
```

```
1 + 2
3 + 3
6 + 4
10 + 5
15
```

Se puede indicar un valor inicial:

```
# valor inicial 0 hace que se aplique la expresión
# f(x,y) x = valor inicial, y = primer elemento
reduce(lambda x,y: x+y**2, [1, 2, 3, 4], 0)
```

Si se especifica valor inicial, se comienza a aplicar la expresión entre el valor inicial y el primer elemento del iterable; si no se especifica, se empieza por el primer par de elementos del iterable.

```
>>> v = [2, 4, 5]
>>> functools.reduce(lambda x,y: x+y**2, v)
43
hace 2 + 4**2 = 18 + 5**2 = 43
```

```
>>> functools.reduce(lambda x,y: x+y**2, v, 0)
45
hace 0 + 2**2 = 4 + 4**2 = 20 + 5**2 = 45
```

En este caso, para una expresión $x + y^2$ el resultado difiere si se especifica o no valor inicial.

Versiones con `reduce()` de algunas funciones conocidas:

```
# versión con reduce() de sum()
sum = lambda iterable: reduce(lambda x, y: x+y, iterable)

# versión con reduce() de count
count = lambda iterable: reduce(lambda x, y: x+1, iterable, 0)

# versión con reduce() de min()
min = lambda iterable: reduce(lambda x, y: x if x < y else y, iterable)
```

Ejemplo:

Supongamos que nos pasan una lista de alturas de personas para obtener su promedio:

```
alturas = ["1,87", "1.65", "1.9", "1,45", "1 72" ]
```

Deberíamos corregirlas para que el separador de decimales sea "." en lugar de una ",", o un " "

```
import functools
import operator
```

```
alturas = ["1,87", "1.65", "1.9", "1,45", "1 72" ]

def normaliza_decimales(x):
    try:
        return float(x)
    except ValueError:
        if " " in x:
            return float(x.replace(" ", "."))
        else:
            return float(x.replace(",", "."))

# calcula sumatoria
suma_alturas = functools.reduce(operator.add, map(normaliza_decimales,
alturas))

# muestra el promedio
print(suma_alturas / len(alturas))
```