

Programación funcional y orden superior

En el modelo de programación funcional, la programación gira por completo en torno al concepto de función. Las funciones son tan naturales como los datos, y juegan el papel de parámetros o resultados de otras funciones con total normalidad; más aún, en los lenguajes funcionales puros no existen variables, y este asunto que puede decirse tan rápidamente tiene consecuencias importantísimas: por ejemplo, no hay instrucción de asignación, no hay bucles. Y muchas otras cosas más.

En esta pequeña sesión exploramos algunos conceptos de la programación funcional que incluye Python.

1. Orden superior

Las funciones pueden jugar el papel de parámetros o resultados de otras funciones, llamadas entonces f.o.s. por funciones de orden superior (*h.o.f.*, por *higher order functions*, en inglés).

```
In [1]: ▶ def cua(n):                # Esta función es de primer orden
          return n*n

          def re_aplica(f, n):      # Esta función es una f.o.s.: un parámetro cuyo
          return f(f(n))

          print(re_aplica(cua, 5))
```

625

```
In [2]: ▶ def elevar_a(n):          # Esta función es una f.o.s.: el resultado es un
          def elevar(x):
              return x ** n
          return elevar

          cubo = elevar_a(3)        # La función cubo es una función de primer orde
          print(cubo(4))
```

64

```
In [3]: ▶ print(elevar_a(3)(4))
```

64

```
In [4]: ▶ def componer(f, g):                                # f.o.s.: dos parámetros son funciones
        def fun_result(x):
            return f(g(x))
        return fun_result

        print(componer(cua, cua)(3))
```

81

- ¿La función `componer(cua, cua)` es de primer orden o de orden superior? Define esa función.

Aplicación: $\int_a^b f(x)dx$

```
In [5]: ▶ def integral(f, a, b):
        n = 100      # num de trozos iguales en que dividimos el intervalo [a,
        suma_acum = 0.0
        """
        Calcular el sumatorio
        """
        return suma_acum

        def f(x):
            return x*x

        def g(x):
            return 2*x - 1

        print(integral(f, 0, 2))
        print(integral(g, 0, 1))
```

0.0

0.0

Ejercicio

Otras funciones conocidas de orden superior que puedes diseñar ahora son las siguientes:

- La derivada de una función (derivable) en un punto
- El método de bipartición para calcular el cero de una función en un intervalo supuesto que...
- Newton-Raphson, partiendo de un punto y supuesto que...

2. Lambda expresiones

La función $x \rightarrow x^2$ convierte un dato `x` en el resultado `x**2`. Observa su manejo en Python:

```
In [6]:  # x -> x**2
        print((lambda x : x**2)(5))
```

25

Imagina la función polinómica $x \rightarrow 2x^2 + 3x + 4$. No ese polinomio, sino esa función polinómica. Puede escribirse y aplicarse así en Python:

```
In [7]:  (lambda x : 2*x*x + 3*x + 4)(2.0)
```

Out[7]: 18.0

La función siguiente toma los parámetros a, b, c y da como resultado esa función polinómica, pero no con los coeficientes 2, 3, 4, sino para los coeficientes a, b, c .

En notación matemática:

$$a, b, c \rightarrow (x \rightarrow ax^2 + bx + c)$$

En Python:

```
In [8]:  fun_pol = lambda a, b, c : (lambda x : a*x*x + b*x + c)
        f = fun_pol(2, 3, 4)
        print(f(4))
        print(fun_pol(1, 0, 1)(2))
```

48

5

Ejercicios

- El factorial es la función que toma un parámetro y... (factorial = lambda n : ...) calcula el producto de los números desde 1 hasta el dato n , dado. ¿Sabrías escribir esto con lambda expresiones?
- Dada una lista de nombres de persona, tenemos una función que selecciona los que tienen una longitud menor o igual a una cantidad, dada. Funciona así, por ejemplo:

```
>>>l = ['Ana', 'Marta', 'Patricia', 'Alba', 'Silvia',
        'Gloria', 'Lara']
>>>short_names(l, 5), short_names(l, 3)
(['Ana', 'Marta', 'Alba', 'Lara'], ['Ana'])
```

Se trata de definir la función short_names así:

```
>>>sort_names = list(lambda ... : filter ...)
```

- Dado el término general de una sucesión a_n de reales (que en realidad es una función $a : \mathbb{N} \rightarrow \mathbb{R}$), podemos dar la lista de términos a_i para $i \in k_1, \dots, k_2$ aplicando la función a a cada uno de los términos de la lista... mediante la función map.
- Expresa la función sumatorio, que suma los términos de una sucesión entre dos límites dados, esto es, $\sum_{i=k_1}^{k_2} a_i$, usando lambda expresiones.

3. Funciones de orden superior predefinidas

La función map

```
In [9]: ▶ # map -----  
  
def mymap(funcion, lista):  
    lista_nueva = []  
    for a in lista:  
        lista_nueva.append(funcion(a))  
    return lista_nueva  
  
lista_a = list(range(10))  
print(lista_a)  
  
# Incremento en una unidad, para pruebas:  
  
def incr_1(n):  
    return n+1  
  
print(mymap(incr_1, lista_a))  
print(map(incr_1, lista_a))  
print(list(map(incr_1, lista_a)))  
  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
<map object at 0x000001BF285B6740>  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pereza

La función map es perezosa: NO se evalúa mientras no haya demanda.

```
In [10]: ▶ lista_a = [1, 2, 3, 4, 5]
lista_b = map(lambda d : 10 / d, lista_a)
print(lista_b)
print(list(lista_b))

print(".....")

# Lo siguiente produciría un error con evaluación impaciente:

lista_a = [5, 4, 3, 2, 1, 0]
lista_b = map(lambda d : 10 / d, lista_a)
print(lista_b)
for e in range(3):
    print(next(lista_b))

<map object at 0x000001BF285B7D30>
[10.0, 5.0, 3.3333333333333335, 2.5, 2.0]
.....
<map object at 0x000001BF285B60B0>
2.0
2.5
3.3333333333333335
```

La función filter

```
In [11]: ▶ # filter -----

def myfilter(predicado, lista):
    lista_nueva = []
    for a in lista:
        if predicado(a):
            lista_nueva.append(a)
    return lista_nueva

lista_a = list(range(10))
print(lista_a)

# La función es_par nos servirá para las pruebas:

def es_par(n):
    return n%2 == 0

print(myfilter(es_par, lista_a))
lista_b = list(filter(es_par, lista_a))
print(lista_b)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
[0, 2, 4, 6, 8]
```

La función filter también es perezosa: NO se evalúa mientras no haya demanda.

La función reduce

```
In [12]: ▶ # reduce -----  
  
def prod(a, b):  
    return a*b  
  
from functools import reduce  
  
lista = range(1, 6)  
print(list(lista))  
  
fact = reduce(prod, range(1, 6))  
print(fact)  
  
[1, 2, 3, 4, 5]  
120
```

Ejercicios

- Define la función *myreduce*, de orden superior... Observa que si la función no fuera asociativa podría funcionar de manera distinta asociando a la derecha o a la izquierda, de manera que es mejor definir dos funciones. Aplícala a la evaluación de polinomios: $\text{eval_pol}(x, [k_0, k_1, \dots, k_n]) = k_0x^n + k_1x^{n-1} + \dots + k_n$.
- Dado el término general de una sucesión a_n de reales (que en realidad es una función $a : \mathbb{N} \rightarrow \mathbb{R}$), podemos dar la lista de términos a_i para $i \in \{k_1, \dots, k_2\}$ aplicando la función a a cada uno de los términos de la lista... mediante la función *map*. También podemos quedarnos con los que cumplen una propiedad y hallar el sumatorio:
$$\sum_{i \in \{k_1, \dots, k_2\}, p(i)} a_i$$

4. Listas intensionales (definidas por comprensión)

```
In [13]: ▶ print([i**2 for i in range(11)]) # una expresión (s  
  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]  
  
In [14]: ▶ print([(i, i**2) for i in range(11)]) # una expresión (u  
  
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49),  
(8, 64), (9, 81), (10, 100)]  
  
In [15]: ▶ print([(i, i**2) for i in range(11) if i%2==0]) # una expresión, u  
  
[(0, 0), (2, 4), (4, 16), (6, 36), (8, 64), (10, 100)]  
  
In [16]: ▶ print([(i, j) for i in range(5) for j in range(3)]) # una expresión y  
  
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2,  
2), (3, 0), (3, 1), (3, 2), (4, 0), (4, 1), (4, 2)]
```

```
In [17]: ▶ print([(i, j) for i in range(10) for j in range(i)]) # una expresión y
```

```
[(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2), (4, 0), (4, 1), (4, 2), (4, 3), (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8)]
```

```
In [18]: ▶ # Generador (perezoso) definido con la notación intensional
```

```
lista = [5, 4, 3, 2, 1, 0]
generador = (10 / d for d in lista)
print(generador)
for e in range(3):
    print(next(generador))
```

```
# OJO: La variable generador es una estructura perezosa:
# Si se evaluara de forma impaciente, daría lugar a un error
# Compruébalo tú mismo:
```

```
# lista_b = [10 / d for d in lista_a]
```

```
<generator object <genexpr> at 0x000001BF28979FF0>
2.0
2.5
3.3333333333333335
```

Ejercicio

- La función `*select_multiplos(n, k)*` genera los números desde 1 hasta n que son múltiplos de k . Defínela usando listas por comprensión.

Otro ejemplo con los números primos

```
In [19]: ▶ import math

def take_until_reach_value(list_in, valor):
    """Pre.: list is long enough, and increasing"""
    i = 0
    list_out = []
    while list_in[i] <= valor:
        list_out.append(list_in[i])
        i = i+1
    return list_out

print(take_until_reach_value(range(1, 1000), math.sqrt(56)))

print("-----")

def es_primo_relativo(n, lista):
    """Dev. True syss n NO es divisible entre ningún entero de la lista"""
    return all(map(lambda d : n % d != 0, lista))

es_primo_relativo(27, [2, 4, 5, 6, 8, 10])

def es_primo(n):
    return n > 1 and es_primo_relativo(n, range(2, math.ceil(math.sqrt(n))))

print(list(filter(es_primo, range(100))))

[1, 2, 3, 4, 5, 6, 7]
-----
[3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Ejercicio

Esta última instrucción

```
print(list(filter(es_primo, range(2, 100))))
```

se puede expresar mediante una lista definida por comprensión:

```
[... for ... if ...]
```

Eso es lo que te pido.

Ejercicio resuelto: Número de vocales

(En este ejercicio ponemos en juego las listas intensionales y las expresiones λ .)

Define una función que cuente el número de vocales de una palabra, usando una expresión lambda:

```
num_vocales = lambda ..... .
```

Por ejemplo la aplicación siguiente dará como resultado un 5:


```
In: num_vocales("murciélago")
```

```
Out: 5
```

```
In [20]: ▶ # Solución:

num_vocales = lambda palabra: sum([1 for letra in palabra if letra in "aeiouy"])

# Comprobación de funcionamiento:

for palabra in ["murciélago", "caramelo", "sol", "supercalifrafic", "luna", "cántaro"]:
    print(palabra, num_vocales(palabra))

murciélago 5
caramelo 4
sol 1
supercalifrafic 6
luna 2
cántaro 3
```

```
In [21]: ▶ # Otra versión del contador de vocales:

num_vocales = lambda palabra: sum([palabra.count(letra) for letra in "aeiouy"])

# Comprobación de funcionamiento:

for palabra in ["murciélago", "caramelo", "sol", "supercalifrafic", "luna", "cántaro"]:
    print(palabra, num_vocales(palabra))

murciélago 5
caramelo 4
sol 1
supercalifrafic 6
luna 2
cántaro 3
```

5. Listas y generadores. Listas infinitas y evaluación perezosa

```
In [22]: ▶ def natural_numbers():
            i = 0
            while True:
                yield i
                i = i+1

            def tomar_n_valores_de(generator, n):
                cont = 0
                for i in generator:
                    yield i
                    cont = cont + 1
                    if cont == n:
                        return

            lista = tomar_n_valores_de(natural_numbers(), 50)
            print(list(lista))
```


```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

```
In [23]: ▶ def tomar_hasta_rebasar(generator, valor):
            for i in generator:
                yield i
                if i > valor:
                    return

            lista = tomar_hasta_rebasar(natural_numbers(), 30.4)

            print(list(lista))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
```

In [24]:  zeta = natural_numbers()

```
for i in zeta:
    print(i, end= " ")
    if i >= 50:
        print()
        break

print("-----")

for i in zeta:
    print(i, end= " ")
    if i > 100:
        print()
        break
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
-----
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98
99 100 101
```

6. Aplicación: el orden superior como parámetro de la ordenación

In [25]: ▶ *# Cosas preliminares:*

```
import random
lista = list(range(0,100)) # todos los números de dos cifras

print(lista)

print("-----")

def sumCifras(n):
    return n//10 + n%10

print(list(map(sumCifras, lista)))

print("-----")

lista.sort(key=sumCifras)
print(lista)

print("-----")
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73,
74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
92, 93, 94, 95, 96, 97, 98, 99]
```

```
-----
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2, 3, 4,
5, 6, 7, 8, 9, 10, 11, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 4, 5, 6, 7, 8,
9, 10, 11, 12, 13, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 6, 7, 8, 9, 10, 1
1, 12, 13, 14, 15, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 8, 9, 10, 11, 1
2, 13, 14, 15, 16, 17, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
```

```
-----
[0, 1, 10, 2, 11, 20, 3, 12, 21, 30, 4, 13, 22, 31, 40, 5, 14, 23, 32,
41, 50, 6, 15, 24, 33, 42, 51, 60, 7, 16, 25, 34, 43, 52, 61, 70, 8, 1
7, 26, 35, 44, 53, 62, 71, 80, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 1
9, 28, 37, 46, 55, 64, 73, 82, 91, 29, 38, 47, 56, 65, 74, 83, 92, 39,
48, 57, 66, 75, 84, 93, 49, 58, 67, 76, 85, 94, 59, 68, 77, 86, 95, 69,
78, 87, 96, 79, 88, 97, 89, 98, 99]
```

Ejercicios

- Genera una lista con 25 pares de números enteros aleatorios, entre 1 y 10: son las coordenadas de 25 puntos del plano discreto. Almacenamos esta lista en una variable *lista_inicial*, y en otra *lista_de_trabajo*, con la que vamos a trabajar.
- Ahora, define una función que reciba dos puntos del plano discreto (dos pares de enteros) y calcule la distancia euclídea entre dichos puntos.
- Define ahora una función de orden superior tal que, dado un punto P , dé la función $dist_P$, que calcula la distancia (a P) de un punto: $dist_P(Q) = ||\overline{PQ}||$.

- Define una función que, dado un punto P y una lista de puntos, devuelve la lista de puntos dada, pero ordenada de menor a mayor distancia a P .

(Nota: chequea que la función `sortad(lista)` da una lista ordenada, pero no cambia la lista