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; no existen variables, y este asunto que puede decirse tan rápidamente tiene consecuencias importantísimas: por ejemplo, no hay bucles. Y muchas otras cosas más.

En esta pequeña sesión exploramos algunos conceptos de la programación funcional que se inculye 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 suyo es una función
    return f(f(n))

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

```
625
```

In [2]:

```
def elevar_a(n):        # Esta funciónes una f.o.s.: el resultado es una función
    def elevar(x):
        return x ** n
    return elevar

cubo = elevar_a(3)       # La función cubo es una función de primer orden.

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 y el resultado también
    def fun_result(x):
        return f(g(x))
    return fun_result

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

# ¿La función "componer(cua, cua)" es de primer orden o de orden superior? Define esa funci
```

```
81
```

Aplicación: $\int_a^b f(x)dx$

In [5]:

```
def f(x):
    return x*x

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

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

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. Funciones de orden superior predefinidas

La función map

In [6]:

```
# 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)

# Definimos una función de primer orden que nos servirá luego:

def incr_1(n):
    return n+1

print(mymap(incr_1, lista_a))
lista_b = list(map(incr_1, lista_a))
print(lista_b)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pereza

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

In [7]:

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

# 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 0x0000019C623605F8>
[10.0, 5.0, 3.3333333333333335, 2.5, 2.0]
<map object at 0x0000019C623606A0>
2.0
2.5
3.3333333333333335
```

La función filter

In [8]:

```
# 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)

# Definimos una función de primer orden que nos servirá luego:

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 [9]:

```
# 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: eval_pol(x, $[k_0, k_1, \ldots, k_n]$) = $k_0 x^n + k_1 x^{n-1} + \ldots + k_n$.

- Dado el término general de una sucesión a_n de reales (que en realidad es una función a : N -> R), podemos dar la lista de términos a_i para $i \in \{k_1, \ldots, k_2\}$ 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: $\sum_{i \in \{k_1, \ldots, k_2\}, p(i)} a_i$

3. Lambda expresiones

In [10]:

```
f = lambda x : x*x

print(f(5))
```

```
25
```

In [11]:

```
print((lambda x : (x+1)*2)(2))     # Usamos la función x -> (x+1)*2, sin nombre, y la aplic
```

```
6
```

In [12]:

```
fun_p2g = lambda a, b, c : (lambda x : a*x*x + b*x + c)
f = fun_p2g(2, 3, 4)
print(f(4))
print(fun_p2g(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', 'Lar
a']
>>>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 : N -> R), podemos dar la lista de términos a_i para $i \in k_1, \ldots, k_2$ 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.
- El sumatorio en sí es una función de orden superior, pues recibe como parámetro una sucesión...

Aplicación: el orden superior como parámetro de la ordenación

In [13]:

```
# 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, 2
1, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 4
0, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 5
9, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 7
8, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 9
7, 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, 11, 12, 13, 14, 1
5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1
7, 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, 5
0, 6, 15, 24, 33, 42, 51, 60, 7, 16, 25, 34, 43, 52, 61, 70, 8, 17, 26, 35,
44, 53, 62, 71, 80, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 19, 28, 37, 46, 5
5, 64, 73, 82, 91, 29, 38, 47, 56, 65, 74, 83, 92, 39, 48, 57, 66, 75, 84, 9
3, 49, 58, 67, 76, 85, 94, 59, 68, 77, 86, 95, 69, 78, 87, 96, 79, 88, 97, 8
9, 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: observa que la función *sorted(lista)* da una lista ordenada, pero no cambia la lista dada.)

4. Listas definidas por comprensión

In [14]:

```
print([i**2 for i in range(11)])                    # una expresión (sencilla) y un gener
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [15]:

```
print([(i, i**2) for i in range(11)])               # una expresión (una tupla) y un gene
```

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 6
4), (9, 81), (10, 100)]
```

In [16]:

```
print([(i, i**2) for i in range(11) if i%2==0])     # una expresión, un generador y un fi
```

```
[(0, 0), (2, 4), (4, 16), (6, 36), (8, 64), (10, 100)]
```

In [17]:

```
print([(i, j) for i in range(5) for j in range(3)])   # una expresión y dos generadores
```

```
[(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 [18]:

```
print([(i, j) for i in range(10) for j in range(i)])  # una expresión y dos generadores, el
```

```
[(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)]
```

Ejercicio

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

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 es_primo_relativo(n, range(2, math.ceil(math.sqrt(n))+1))

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

```
[1, 2, 3, 4, 5, 6, 7]
-----------------------------------------------
[0, 1, 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.

5. Listas y generadores. Listas infinitas y evaluación perezosa

In [20]:

```
def natural_numbers():
    i = 0
    while True:
        yield i
        i = i+1

def tomar_n_valores_de(generador, n):
    cont = 0
    for i in generador:
        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, 2
1, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 4
0, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

In [21]:

```
def tomar_hasta_rebasar(generador, valor):
    for i in generador:
        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, 2
1, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
```

In [22]:

```
zeta = natural_numbers()

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

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

for i in zeta:
    print(i, end= " ")
    if i > 200:
        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 5
4 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
-----------------------------------
101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138
139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157
158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176
177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195
196 197 198 199 200 201
```