

# Programación funcional con Python

## Paradigma funcional

El paradigma funcional se basa en el modelo matemático de composición funcional.

Un programa funcional consiste en un conjunto ordenado de declaraciones, que **expresan qué hacer**, en las que no existe el concepto de asignación, como tampoco algoritmos, sentencias, comandos ni estructuras de control de naturaleza imperativa. La importancia del concepto de declaratividad en este paradigma, permite encuadrarlo dentro de los paradigmas declarativos .

La iteración (ej. **while**, **for**) se reemplaza con recursión que es la forma funcional de repetir varias veces una acción; se usan estructuras inmutables: una vez que asignamos algo, no se cambia, para evitar efectos secundarios; no existe el concepto de "variable"

El lenguaje se encarga en forma transparente para el usuario de realizar las evaluaciones necesarias para lograr los resultados deseados, a diferencia de otros paradigmas imperativos en los que el programa describe en detalle cómo hacer algo, los pasos que la máquina debe ejecutar para realizar una tarea.

Los programas funcionales, por lo tanto son más fáciles de diseñar, de escribir y de mantener, testear, pero permiten menos control sobre la máquina (justamente por abstraernos del "cómo hacer")

Algunas características importantes de los lenguajes funcionales:

- **Funciones Puras** - no tienen efectos secundarios / colaterales, o sea que no alteran el estado del programa que las invoca. Y para la misma entrada siempre retornan la misma salida.
- **Inmutabilidad** - los datos no pueden cambiar una vez que son creados.
- **Funciones de orden superior** - Pueden aceptar funciones como argumentos y/o devolver funciones.

Ejemplos:

Una planilla de cálculo es una experiencia similar a la programación funcional. En una hoja se especifica el valor de cada celda en términos de valores de otras celdas. Se expresa qué se necesita, no cómo lograrlo. En general, se especifica el valor de una celda mediante una expresión cuyas piezas se puedan evaluar en cualquier orden, antes que por una secuencia de comandos que compute su valor.

Una consecuencia interesante de la falta de especificación del orden de ejecución de las fórmulas en las planillas de cálculo, es que la noción de la asignación no es muy útil, ya que si no se sabe exactamente cuándo sucederá una asignación, no se puede hacer mucho uso de ella.

Esto la une al paradigma con el que comparte la centralidad de la declaratividad y lo pone en un contraste fuerte con programas en los lenguajes de paradigmas procedurales, que consisten esencialmente en una secuencia cuidadosamente especificada de asignaciones en la cual el orden de las instrucciones y asignaciones es crucial para el significado de un programa.

Este foco en el de alto nivel, de especificar el "**qué**" antes que en el menor nivel del "**cómo**" es una característica para distinguir de lenguajes de programación funcionales.

Otro lenguaje con fuertes rasgos funcionales bien conocido es el lenguaje de consulta estándar de base de datos SQL. Una consulta del SQL es una expresión que implica las proyecciones, selecciones, ensambles y así sucesivamente. Un `select` por ejemplo, especifica qué relación debe ser computada, sin especificar cómo realizarla.

Python se utiliza habitualmente en forma imperativa pero si se desea se puede usar el estilo declarativo.

## Funciones puras

La programación funcional está basada en **funciones puras**: funciones en las que el valor que devuelven depende sólo de sus argumentos (transparencia referencial). No tienen asignaciones destructivas ni efectos secundarios (side effects) es decir, no alteran elementos fuera de la función.

Se comportan como las funciones matemáticas: a igual argumento devuelven el mismo valor; por lo que una vez que se evaluó la función para un valor determinado, este se puede almacenar y utilizar las siguientes veces que se invoque la función con ese argumento en lugar de invocar repetidas veces a la misma función con el mismo argumento: permiten mejorar la eficiencia.

Como ejemplo, en:

```
import math

v = [220, 110, 24, 12, 6]
vp = []
```

```
for valor in v:
    vp.append(valor * math.sqrt(2))

print(vp)
```

el valor de `math.sqrt(2)` se calcula 5 veces, uno por cada elemento de `v`

Los lenguajes funcionales calculan este valor una vez y lo usan las veces que sea necesario.

Ejemplos de función pura ....

```
def pura(x, y):
    temp = x + 2 * y
    return temp / (2 * x + y)
```

...y una función "impura" donde hay interacción con elementos fuera de la función:

```
valores = []
def impura(n):
    valores.append(n)    # altera la lista valores, que no pertenece al
                        # ámbito de la función
```

## Funciones de orden superior

Como dijimos, el paradigma se basa en el modelo matemático de composición funcional. Por eso, uno de los elementos característicos de la programación funcional son las funciones de orden superior (Higher order functions)

Son funciones que **pueden recibir otras funciones** como argumentos **y/o devolverlas** como resultados.

```
def sumatoria(nums):
    return sum(nums)

def accion(func, numbers):
    return func(numbers)
```

```
print(accion(sumatoria, (1, 2, 3))) # a accion le pasamos sumatoria como
                                     # argumento
```

## map y filter

**Son funciones de orden superior** incorporadas en Python

Reciben como argumentos una función, y un iterable, sobre el que operan.

**map** devuelve un iterable que resulta de aplicar la función que recibe como primer argumento sobre cada uno de los elementos del iterable que recibe como segundo argumento

```
def sumar_5(x):
    return x + 5

nums = [11, 22, 33, 44, 55]

lista_nueva = list(map(sumar_5, nums)) # suma 5 a todos los elementos
print(lista_nueva)
```

Devuelve [16, 27, 38, 49, 60]

Es equivalente a hacer:

```
lista_nueva = []
for p in nums:
    lista_nueva.append(p + 5)

print(lista_nueva)
```

**filter** devuelve un iterable formado por los elementos del iterable que recibe como segundo argumento para los cuales la función recibida como primer parámetro sea verdadera

```
def esPar(x):
    return x % 2 == 0    # True si x es par

nums = [11, 22, 33, 44, 55]
res = list(filter(esPar, nums))
print(res)
```

Devuelve [22, 44]

Es equivalente a hacer:

```
res = []
for p in nums:
```

```

    if p % 2 == 0:
        res.append(p)

print(res)

```

map y filter nos evitan tener que iterar con `for n in iterable`

## Funciones anónimas

En los ejemplos de map y filter usamos funciones (sumar\_5, esPar) definidas con def en la forma habitual.

Existe la posibilidad de usar funciones sin necesidad de declararlas previamente lo que permite la flexibilidad de escribir funciones como una expresión cualquiera, muy útil cuando queremos pasar funciones como argumentos... esto es posible usando funciones anónimas.

Para expresar una función anónima usamos lambda seguida de los parámetros, luego : y luego la expresión que devuelven.

```
lambda p: p * 2
```

esto es equivalente a

```
def funcion_anonima(p):
    return p * 2
```

otro:

```
def funcion(f, arg):
    return f(arg)          # devuelve el resultado de evaluar f
                           # con el argumento arg
```

```
my_func(lambda x: x*x, 5)
```

En `lambda x: x*x` La función recibe un argumento x y devuelve el argumento al cuadrado. Notar que las funciones anónimas sólo podrán realizar tareas que requieran una única expresión.

```
# función con nombre
def poly(x):
    return x**2 + 5*x + 4

print(poly(-4))
```

```
# función anónima
print((lambda x: x**2 + 5*x + 4) (-4))
```

Una función anónima se puede asignar a una variable y usarla como una función “común”.

```
darElDoble = lambda x: x * 2
print(darElDoble(7))
```

En los ejemplos con map y filter podemos aplicar lambda para expresar las operaciones de forma más compacta:

```
nums = [11, 22, 33, 44, 55]

lista_nueva = list(map(lambda n: n + 5, nums)) # suma 5 a todos los
                                                # elementos

res = list(filter(lambda m: m % 2 == 0, nums)) # devuelve solo pares
```

## Generadores

Son iterables y como tales se pueden recorrer secuencialmente (con `for n in iterable`) No permiten hacer referencia a un elemento usando su índice (Ej. `gene[9]` no está permitido si `gene` es un generador)

En lugar de ocupar la memoria con la totalidad de una secuencia, los generadores **calculan el siguiente elemento** del iterable y lo devuelven con **yield** (en lugar de `return`)

El estado de la función se mantiene entre sucesivas llamadas. **yield** “convierte” una función en generador: los objetos dentro de la función no se destruyen entre llamados, mantienen su valor.

```
def genVocales():
    v = ['a', 'e', 'i', 'o', 'u']
    for l in v:
        yield l

for letra in genVocales():
    print(letra)
```

```
muestra:
a
```

e  
i  
o  
u

```
def lanzamiento():  
    i = 4      # a i se le asigna 4 solo la primera vez que se ejecuta el  
               # generador.  
               # los cambios en su valor se mantienen en sucesivas llamadas.  
    while i > 0:  
        yield i  
        i -= 1  
  
for i in lanzamiento():  
    print(i)
```

4  
3  
2  
1

Al devolver un elemento por vez, los generadores **no necesitan memoria para guardar toda una serie de valores** como sería necesario en una lista. Por esto, se pueden hacer **generadores infinitos....**

```
def infinitasX():  
    x = 0  
    while True:  
        x += 1  
        yield x  
  
for i in infinitasX():  
    print(i)
```

1  
2  
3  
etc.

Los generadores, siempre que sean finitos, pueden transformarse en listas, tuplas, etc. usando el constructor apropiado: `list()`, `tuple()`, etc.

```
def numerosParesHasta(x):  
    for i in range(x):  
        if i % 2 == 0:
```

```
yield i
```

```
print(tuple(numerosParesHasta(11)))
```

```
(0, 2, 4, 6, 8, 10)
```

Los generadores mejoran la eficiencia porque utilizan generación a demanda (on demand, **lazy**) esto es un valor por vez, lo que reduce el uso de memoria. Además se puede usar el primer valor generado sin necesidad de esperar que se genere primero todo el conjunto.

En versiones 2.x de Python, `range(n)` se desarrollaba en memoria antes de usarlo... esto podría generar excepciones por falta de memoria en caso de usar valores muy grandes de `n`. Para estos casos donde se necesitaba usar `range()` como generador existía `xrange(n)`.

En Python 3.x `range()` funciona como generador, como el `xrange()` en 2.x.

## Comprensión de listas

Es una forma compacta de generar listas (u otros iterables) con Python.

```
[función for item in iterable]
```

Como ejemplo, para generar una lista de los primeros 10 números naturales al cuadrado:

```
lista = [p ** 2 for p in range(10)]
```

Con esto se obtiene el mismo resultado que con:

```
lista = list(map(lambda p: p** 2, range(10)))
```

Si quisiéramos generar una lista solo con los números pares de otra lista:

```
nums = [11, 22, 33, 44, 55]
```

```
lista_nueva = [n for n in nums if n % 2 == 0]
```

El resultado es equivalente a usar `filter`:

```
lista_nueva = list(filter(lambda m: m % 2 == 0, nums))
```

Para elevar al cuadrado cada elemento negativo de una lista, y dejar como están los positivos o cero::

```
lista_nueva = [n ** 2 if n < 0 else n for n in lista] # el if pasa  
# adelante :(
```

Para aumentar 10 % a los valores mayores a 300 y aumentar 20% al resto:

```
lista = [500, 100, 400, 200]  
lista_nueva = [n * 1.1 if n > 300 else n * 1.2 for n in lista]  
  
print(lista_nueva)
```

mostrará: [550.0, 120.0, 440.00000000000006, 240.0]

También se pueden generar diccionarios:

```
palabras = ('auto', 'casa', 'barco')
```

```
hist = {pal: len(pal) for pal in palabras}
```

```
print(hist)
```

mostrará {'auto': 4, 'casa': 4, 'barco': 5}

```
dicc = {i:("Even" if i%2==0 else "Odd") for i in range(10)}
```

```
{0: 'Even', 1: 'Odd', 2: 'Even', 3: 'Odd', 4: 'Even', 5: 'Odd', 6:  
'Even', 7: 'Odd', 8: 'Even', 9: 'Odd'}
```

Video (hasta min. 28) <https://www.youtube.com/watch?v=eqtVRsdIL3s>