

Programación Funcional con Python

Imperativa vs Declarativa:

Imperativa

```
numbers = [18, 6, 4, 15, 55, 78, 12, 9, 8]

counter = 0
for number in numbers:
    if number > 10:
        counter += 1

print(counter)
# 5
```

Declarativa

```
numbers = [18, 6, 4, 15, 55, 78, 12, 9, 8]
counter = len(list(filter(lambda num: num > 10, numbers)))

print(counter)
# 5
```

Funciones Puras:

Dada una entrada siempre se espera la misma salida sin afectar valores externos o requerir de otras variables.

```
def multiply_by_2(x):
    return x * 2

multiply_by_2(3) # 6
multiply_by_2(3) # 6
multiply_by_2(3) # 6
```

Funciones impuras:

- Dado una entrada no se puede determinar la salida

```
from random import randint

def multiply(x):
    return x * randint(1, 10)

multiply(3) # 3
multiply(3) # 15
multiply(3) # 27
```

- Su resultado depende de variables externas

```
external_variable = ["Hola", "mundo"]

def custom_join():
    new_string = ""
    for word in external_variable:
        new_string += word
    return new_string

custom_join()
# 'Holamundo'
```

- Su ejecución genera efectos secundarios o side effects modificando el estado de otras variables

```
external_variable = ["Hola", "mundo"]

def remove_last_value():
    external_variable.pop()

print(external_variable)
# ['Hola', 'mundo']

remove_last_value()

print(external_variable)
# ['Hola']
```

Funciones como ciudadanos de primer orden:

```
def my_function():
    print("I am function my_function()!")
```

- Ser asignados a variables:

```
my_function()
# I am function my_function()!

another_name = my_function
another_name()
# I am function my_function()
```

- Almacenados en estructuras de datos:

```
objects = ["cat", my_function, 42]
print(objects[1])
# <function my_function at 0x10aad2020>

dictionary = {"cat": 1, my_function: 2, 42: 3}
dictionary[my_function]
# 2
```

- Pasados como argumento a funciones:

```
def inner():
    print("I am function inner()!")

def outer(function_param):
    function_param()

outer(inner)
# I am function inner()!
```

Nota: Python proporciona una notación abreviada llamada decorador para facilitar el wrapping de una función dentro de otra.

- Devueltos como resultado de funciones:

```
def outer():
    def inner():
        print("I am function inner()!")
    return inner

returned_function = outer()
returned_function()
# print("I am function inner()!")

print(returned_function)
# <function outer.<locals>.inner at 0x10aad2200>
```

Funciones de orden superior

Ejemplo

```
def inner():
    print("I am function called by a High order function!")

def high_order_function(function_param):
    function_param()

high_order_function(inner)
# I am function called by a High order function!
```

Función Filter como Función de orden superior

```
def get_even(x):
    return x%2 == 0

integer = range(100)
even = list(filter(get_even, integer))
print(even)
# [0, 2, 4, 8..., 98]
```

Recursion en Python

```
def countdown(n):
    print(n)
    if n > 0:
        countdown(n - 1)

countdown(5)
# 5
# 4
# 3
# 2
# 1
# 0
```

Opción no recursiva

```
def countdown(n):
    while n >= 0:
        print(n)
```

```

n -= 1

countdown(5)
# 5
# 4
# 3
# 2
# 1
# 0

```

Útiles para búsqueda y ordenamiento de datos:

```

def buscar_en_arbol(nodo, valor):
    if nodo is None:
        return False
    if nodo.valor == valor:
        return True
    return buscar_en_arbol(nodo.izq, valor) or buscar_en_arbol(nodo.der,
valor)

```

Currying

```

# Currying manual
def suma_curried(a):
    def suma_b(b):
        def suma_c(c):
            return a + b + c
        return suma_c
    return suma_b

# Usamos la función curried paso a paso
resultado = suma_curried(1)(2)(3) # Suma: 1 + 2 + 3 = 6
print(resultado) # Resultado: 6

```

Funciones Lambda

```

def get_even(x):
    return x%2 == 0

get_even
# <function get_even at 0x10aad2340>

```

```
lambda x: x%2 == 0
# <function <lambda> at 0x10aad2480>
```

- Una función Lambda es callable (llamable o invocable).

```
callable(lambda s: s[::-1])
# True
```

- Una función lambda es asignable a variables.

```
def reverse(s):
    return s[::-1]

reverse("I am a string")
# gnirts a ma I

lambda_reverse = lambda s: s[::-1]
lambda_reverse("I am also a string")
# 'gnirts a osla ma I'
```

- Una función lambda puede ser llamada inmediatamente (IIFE).

```
(lambda s: s[::-1])("I am a string")
# 'gnirts a ma I'
```

- Una función lambda puede recibir multiples parametros.

```
lambda_average = lambda x1, x2, x3: (x1 + x2 + x3) / 3

lambda_average(3, 6, 9)
# 6.0
```

- Una función lambda puede no recibir parametros.

```
meaning_of_life = lambda: 42

meaning_of_life()
# 42
```

- Una función lambda pueden retornar multiples resultados en una tupla pero solo de forma explícita.

```
def my_function(x):
    return x, x ** 2, x ** 3

my_function(3)
# (3, 9, 27)

(lambda x: x, x ** 2, x ** 3)(3)
# <stdin>:1: SyntaxWarning: 'tuple' object is not callable; perhaps you
# missed a comma?
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# NameError: name 'x' is not defined

(lambda x: (x, x ** 2, x ** 3))(3)
# (3, 9, 27)
```

- Una función lambda puede retornar una lista o un diccionario

```
(lambda x: [x, x ** 2, x ** 3])(3)

(lambda x: {1: x, 2: x ** 2, 3: x ** 3})(3)
```

- Una función lambda puede ser creada dentro de un f-string pero debe ser encerrada dentro de paréntesis.

```
print(f"--- {lambda s: s[::-1]} ---")
# File "<stdin>", line 1
#   (lambda s)
#           ^
# SyntaxError: f-string: invalid syntax

print(f"--- {(lambda s: s[::-1])} ---")
# --- <function <lambda> at 0x10aad2520> ---

print(f"--- {(lambda s: s[::-1])('I am a string')} ---")
# --- gnirts a ma I ---
```

Función Map

Sintaxis `map(<f>, <iterable>)`

```
def reverse(s):
    return s[::-1]

reverse("I am a string")
```

```
# 'gnirts a ma I'

animals = ["cat", "dog", "bird", "gecko"]
iterator = map(reverse, animals)
print(iterator)
# <map object at 0x7fd3558cbef0>

for i in iterator:
    print(i)
# tac
# god
# drib
```

- Si el iterable contiene elementos que no son adecuados para la función especificada, Python genera una excepción:

```
list(map(lambda s: s[::-1], ["cat", "dog", 3.14159, "gecko"]))
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
#   File "<stdin>", line 1, in <lambda>
# TypeError: 'float' object is not subscriptable

"+" .join([1, 2, 3, 4, 5])
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: sequence item 0: expected str instance, int found

"+" .join(map(str, [1, 2, 3, 4, 5]))
# '1+2+3+4+5'
```

- Al poder aplicarse a cualquier iterador puede ser utilizado en strings.

```
hello_world = 'hello world'
iterator = map(lambda x: x.upper(), hello_world)
print(''.join(iterator))
# HELLO WORLD

print(hello_world)
# hello world
```

Llamando map() con multiples Iterables

Sintaxis `map(<f>, <iterable1>, <iterable2>, ..., <iterablen>)`

```
def f(a, b, c):
    return a + b + c
```



```
list(map(f, [1, 2, 3], [10, 20, 30], [100, 200, 300]))
# [111, 222, 333]
```

map(f, [1, 2, 3], [10, 20, 30], [100, 200, 300])

[f(1, 10, 100), f(2, 20, 200), f(3, 30, 300)]

[111, 222, 333]

Functional Programming in Python: When and How to Use It – Real Python

Creando el mismo resultado con una función Lambda

```
list(
    map(
        (lambda a, b, c: a + b + c),
        [1, 2, 3],
        [10, 20, 30],
        [100, 200, 300]
    )
)
```

Función Filter

Sintaxis `filter(<f>, <iterable>)`

```
def greater_than_100(x):
    return x > 100

list(filter(greater_than_100, [1, 111, 2, 222, 3, 333]))
# [111, 222, 333]
```

Se puede crear la misma expresión con una función lambda

```
list(filter(lambda x: x > 100, [1, 111, 2, 222, 3, 333]))  
# [111, 222, 333]
```

Ya que la función Range genera un iterable de números de 0 a n-1 también puede ser pasado a la función Filter

```
def is_even(x):  
    return x % 2 == 0  
  
list(filter(is_even, range(10)))  
# [0, 2, 4, 6, 8]  
  
list(filter(lambda x: x % 2 == 0, range(10)))  
# [0, 2, 4, 6, 8]
```