1. Generalidad

August 31, 2021

1 Que significa que python es multiparadigma?

python es un lenguaje interpretado y ademas es tipado, se trata de unlenguaje de programación multiparadigma, ya que soporta:

- Programación imperativa
- Programación orientada a objetos
- programación funcional

es decir que acepta diversas formas de trabajar con el lenguaje.

[1]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.1 Definiciones:

- Programación imperativa (estructurada), el código será ejecutado desde el principio del fichero al final sin seguir ningún tipo de desviación. Su mayor ventaja radica en su simplicidad y

poco peso. Su peligrosidad es el código espagueti, archivos con centenares o miles de líneas donde solo unos pocos seres humanos son capaces de modificar y salir victoriosos.

- Programación orientada a objetos (OOP o Object Oriented Programming), donde se encapsulan las variables y funciones en pequeños módulos capaces de clonarse y modificarse. Su punto fuerte es la capacidad de re-utilización y aislamiento para evitar problemas con otras funcionalidades. La parte negativa recae en la complejidad de crear buenos objetos y la depuración.
- Programación funcional (FP o Functional programming), donde el código se reparte en sencillas funciones capaces de ser invocadas con variables u otras funciones. Su facilidad de uso por atomicidad logra un mantenimiento sólido y compatible con casi cualquier lenguaje. Además su inmutabilidad de variables evita gran parte de los problemas que si sufre la programación orientada a objetos.

1.2 Principios de la programación funcional

- Uso de funciones: Como su nombre indica, todo se construye por medio de funciones. Esta forma de trabajar no solo es sencilla, ordenada, clara, fácil de testear, sino que además es una práctica que han utilizado grandes figuras militares como Julio César y Napoleón. No, no usaron Python (al menos no hay constancia de ello), pero aplicaron el concepto de: **"divide y vencerás"**. Y la programación funcional usa esta estrategia para prácticamente todo.
- Funciones de primera clase: Las funciones son tratadas como una variable más. Incluso pueden ser devueltas.
- Funciones puras: Totalmente predictivo, los mismos datos de entrada producirán los mismos datos de salida. Puedes sustituir el parámetro de entrada de sin que ello altere el flujo del programa.
- Recursividad: Las funciones se pueden llamar a si mismas simplificando tareas como recorrer árboles de datos o la gestión de bucles controlados.
- Inmutabilidad: No hay variables, solo constantes. Personalmente comprender su potencial y llevarlo a la práctica fue como darle al reset de mi cerebro; tuve que re-aprender a usar variable. Anécdota a parte; ¿donde suele fallar el software? En gran mayoría de las ocasiones viene por una variable que a sido cambiada. Esto provoca que un bloque de código se ejecute con unas condiciones no previstas por nadie en el mundo mundial, toca revisar cada variable en diferentes valores hasta que encontramos al culpable. Os hago una reflexión: ¿Y si esas variables nunca fueran modificadas? O siendo más prácticos, ¿y si creamos una nueva constante de cada modificación? ¿Y si… os digo que a nivel de rendimiento… es más eficiente? Es un concepto muy interesante de aplicar.
- Evaluación perezosa (no estricta): En la programación funcional podemos trabajar con expresiones que no han sido evaluadas, o dicho de otra manera, podemos disponer de variables con operaciones cuyo resultado aún no se conoce. A esto se le denomina evaluación no estricta. Un efecto secundario es el aumento de rendimiento, y otra es que podemos realizar locuras como hacer cálculos con operaciones muy complejas o listas infinitas sin realizar calculos. ¿Cómo es esto posible? Porque se trabaja con expresiones matemáticas, solo se calcula el valor cuando lo necesitas como por ejemplo al realizar un print.

1.3 Lambda

son pequeñas funciones anonimas, restrictivas y concisas.

lambda(function, list)

[34]:
```
def incrementar(x):
    return x + 1

incrementar(9)
```

[34]: 10

[35]:
```
(lambda x: x + 1)(9)
```

[35]: 10

[36]:
```
incremento = lambda x: x + 1

incremento(9)
```

[36]: 10

2 Closure

Funciones dentro de otras funciones.

[37]:
```
# Función que devuelve una función
def construir_multiplos(factor):
    def interno(valor):
        return valor * factor
    return interno

# Guardamos la expresión
multiplos_de_2 = construir_multiplos(2)
multiplos_de_7 = construir_multiplos(7)

# Evaluamos

multiplos_de_2(10)


multiplos_de_7(2)
```

[37]: 14

2.1 Filter

Su objetivo es convertir un elemento iterable (como una tupla o lista), en otra pero de igual o inferior tamaño; filtrando por lo que quieras.

filter(funcion, elemento_iterable)

[38]:
```
lista = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

def mayor_que(valor: list):
    if  valor > 5:
        return True
    else:
        pass

list(filter(mayor_que, lista))
```

[38]:
```
[6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

[39]:
```
list(filter(lambda x: x > 5, lista))
```

[39]:
```
[6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

2.2 map()

En este caso se utiliza para modificar el valor de una secuencia siguiendo una regla.

map(funcion, elemento_iterable)

[40]:
```
def doblar(numero):
    return numero*2

numeros = [2, 5, 10, 23, 50, 33]

list(map(doblar, numeros))
tuple(map(doblar, numeros))
```

[40]:
```
(4, 10, 20, 46, 100, 66)
```

2.3 reduce()

Su estructura costa de 3 argumentos: la expresión, el elemento que deseamos recorrer y la variable de inicio.

La podemos encontrar dentro de la librería functools.

functools.reduce(funcion, elemento_iterable, variable_inicial)

[31]:
```
def suma(a, b):
    result = a + b
    print(f"{a} + {b} = {result}")
```

```
    return result
numbers = [1, 2, 3, 4]
reduce(suma, numbers)
```

```
1 + 2 = 3
3 + 3 = 6
6 + 4 = 10
```

[31]: 10

[32]:
```
from functools import reduce

numbers = [1, 2, 3, 4]

reduce(lambda a,b: a*b, numbers)
```

[32]: 24