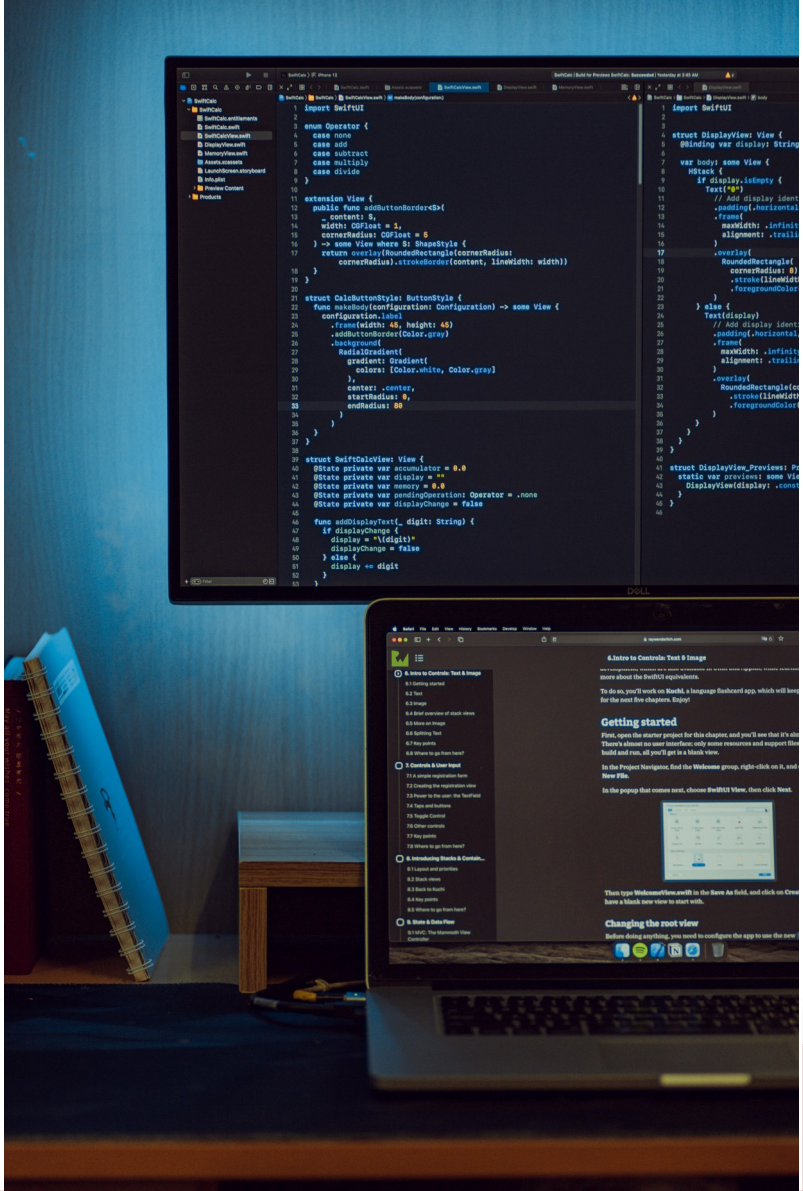
Curso Python

Escuela Politécnica Superior

Grado Ingeniería del automóvil - Curso 22/23

Conceptos básicos



UNIVERSIDAD NEBRIJA

Profesor:
José Ignacio Naranjo
jnaranjo@nebrija.es

Syntax error

Todo lenguaje, compilado o interpretado, tiene unas reglas sintácticas que hay que seguir.

Si no las sigues, el intérprete, en el caso de python, no entenderá las órdenes y devolverá un error, en el caso de que no hayas seguido las normas del lenguaje será un SyntaxError

```
def hola_mundo()
    print("hola mundo")
```

```
  File "/home/josnar/Development/
    def hola_mundo()
                    ^
SyntaxError: expected ':'
```

Indentation error

Python se estructura mediante indentaciones. Nada de llaves como otros lenguajes. Cada ámbito se representa por una variación en la indentación. Un error en la indentación no permite el inicio de la ejecución del programa.

Los nuevos ámbitos vienen marcados por:

- Funciones
- Bifurcaciones (if/else)
- Loops

```
/home/josnar/Development/Nebrija/curso_
  File "/home/josnar/Development/Nebrij
    a == 0
IndentationError: unexpected indent
```

Datos en un programa

Los datos en un programa se almacenan en posiciones de memoria gestionadas por el SO.

Las dos principales características de un dato son su mutabilidad su ámbito de uso.

- Un dato puede ser constante o variable
- ¿Desde dónde se accesible un dato?

Mutabilidad de una variable

Según su mutabilidad los datos pueden ser:

- Constante: Su valor permanecerá inalterado durante la ejecución de un programa. Si el programa intenta modificar su valor, ocurrirá un error:

  Python diversión 1: No existen las constantes en Python. Solamente existen variables. Existe la convención de que si queremos declarar una constante, lo indiquemos poniendo su nombre en mayúsculas. Ejemplo. PI = 3.1415...

- Variable: Su valor podrá ser modificado durante la ejecución de un programa

  Python diversión 2: Más adelante veremos que las variables en Python tampoco son exactamente una posición de memoria con un valor modificable :)

Ámbito de una variable

Es la región del código desde la que una variable es accesible.

Las regiones de un código funcionan como capas. Primero hay una región global, dentro de esa región hay funciones y cada una será una región. Dentro de una función hay, por ejemplo, un bucle for, que marcará una nueva región dentro de la región de la función.

```
# Global scope
x = 0

def outer():
    # Enclosed scope
    x = 1
    def inner():
        # Local scope
        x = 2
```

Ámbito de una variable II

Python diversión 3: El intérprete de python es imaginativo con los ámbitos de las variables. Por ejemplo, si hay una variable declarada en un if/else o en un bucle, el intérprete la inicializa y la podrás usar fuera de su ámbito original. Esto formalmente no es correcto y puede dar lugar a errores lógicos en tu programa.

```
def scope_test_meh(variable1):
    if len(variable1) > 0:
        name = "Sandra"
    else:
        name = None

    print(f"{variable1} {name}")
```

```
def scope_test_should_fail(variable1):
    if len(variable1) > 0:
        name = "Sandra"

    print(f"{variable1} {name}")  # We
```

```
def scope_test_formally_ok(variable1):
    name = None
    if len(variable1) > 0:
        name = "Sandra"

    print(f"{variable1} {name}")  # All
```

Ámbito de una variable III

Al menos entre funciones sí hay protección en el ámbito

```python
def scope_test1(variable1):
    print(variable1)


def scope_test2(variable2):
    print(variable2)
    print(variable1)


if __name__ == "__main__":
    scope_test1("Juan")
    scope_test2("Luisa")
```

```
Traceback (most recent call last):
  File "/home/josnar/Development/Nebrija/curso_python/curso_python_sesion_1/variables/scope_3.py", line 12, in <module>
    scope_test2("Luisa")
  File "/home/josnar/Development/Nebrija/curso_python/curso_python_sesion_1/variables/scope_3.py", line 7, in scope_test2
    print(variable1)
NameError: name 'variable1' is not defined. Did you mean: 'variable2'?

Process finished with exit code 1
```

Declaración y definición de una variable

En Python la declaración indica la creación de una nueva variable. Debido a que Python es un lenguaje de tipado dinámico, podemos re-declarar la variable (NO VAMOS A HACER TONTERÍAS)

En Python no se informa del tipo de dato de la variable explícitamente, sino que se obtiene implítamente del valor otorgado en la definición.

```
variable_1 = 1
print(variable_1)
variable_1 = "hola"
print(variable_1)
```

Asignación de valores a una variable

Se usa el operador asignación ‘=’. Es decir mi_variable = 0 declara una nueva variable con ese nombre y la define con valor 0.

En muchos lenguajes hay que declarar el tipo que tendrá la variable -ya sea un booleando, un entero, un decimal, un caracter, etc.-. En Python no hace falta declarar el tipo.

El intérprete lo deduce del tipo de dato que entra -y además, como es un lenguaje con tipado dinámico, podemos cambiar el tipo de la variable cuando queramos. Como es una mala práctica, esta parte la olvidamos-

Para actualizar un valor se usa el mismo operador ‘=’. Por ejemplo:

contador = contador + 1

contador += 1

Tipos básicos I (numéricos)

int → Números enteros (-inf, +inf)

Normalmente los enteros en cualquier lenguaje tienen un rango marcado por el número de bits que se usen para representarlo (8 bits, 16bits, 32bits, 64bits). En python no existe esa limitación y se puede representar cualquier número entero. Hay 4 tipos de representaciones (binario, octal, decimal y hexadecimal)

- **float** → Números decimales (-inf, +inf). En este caso sí hay límite del rango de representación, pero es tan grande que vamos a tomarlo como infinito también.
- complex → Número complejos. No los usaremos ni los tendremos en cuenta.

Tipos básicos II

- **bool** → True/False

  El tipo resultado de las operaciones booleanas. Usado en las bifurcaciones (if/elif).

Tipos estructurados I

https://docs.python.org/3/tutorial/datastructures.html

- **list** → Lista de elementos con tipo básico → notas_curso = [7.1, 4.3, 8.5, 5.0]
  - notas_curso.append(3.5) → añade un elemento a la lista
  - notas_curso.clear() → vacía la lista
  - len(notas_curso) → retorna el número de elementos en la lista
  - notas_curso[0] → retorna el primer elemento
  - List slicing → notas_curso[1:] → retorna todos los elementos de la lista menos el primero

Error común a depurar:

```
Traceback (most recent call last):
  File "/home/josnar/Development/Nebr
    print(grades[0])  # IndexError!!
IndexError: list index out of range
```

Tipos estructurados II

- **str** → Cadenas de caracteres. Funcionan como el structurado list → ‘Una cadena de caracteres de cualquier longitud’

  Operaciones básicas sobre cadenas de caracteres.

  - Imprimir por pantalla → print(“hola”)
  - Concatenación → Con el operador + → saludo = “hola, “ + nombre
  - Sustitución de variables (f-strings) → saludo = f’Hola, {nombre}’
  - Partir cadenas de caracteres. “Carlos,Luisa,Pedro,Carmen”.split(‘,’)
  - https://docs.python.org/3/library/stdtypes.html#str

Tipos estructurados II

- **set** → Similar a **list** pero no permite elementos repetidos

  https://docs.python.org/3/library/stdtypes.html#set
- **dict** → Colección de elementos accesible mediante una clave.
  - La clave puede ser de cualquier tipo, aunque lo más común pueda ser un string
  - El valor puede tener cualquier tipo. Un tipo básico u otro tipo estructurado.

Nuevo error típico: KeyError

```
Traceback (most recent call last):
  File "/home/josnar/Development/Nebrij
    print(test_dict["mi_nueva_clave"])
KeyError: 'mi_nueva_clave'
```

Tipos enumerados

Conjunto de valores simbólicos agrupados. Por ejemplo, queremos representar el resultado en modo quiniela de un partido.

Podemos usar strings. Cuando cada el de casa el resultado sería ‘1’, y ‘X’ y ‘2’ en los otros casos. Propenso a errores porque los strings pueden tener muchos más valores.

Podemos usar enteros. 1 cuando gana el de casa, 2 cuando gana el de fuera y 0 en caso de empate. Mismo problema

Hagamos esto:

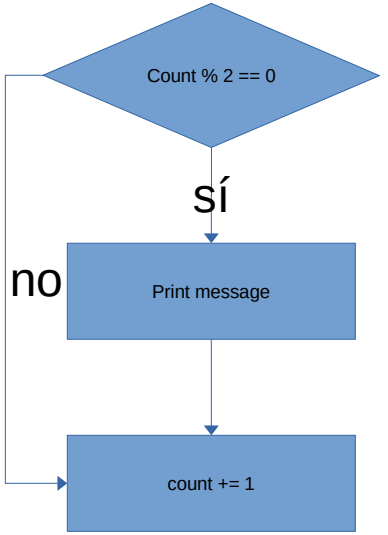
```
class ResultadoQuiniela(Enum):
    HOME_WINS = 0
    DRAW = 1
    AWAY_WINS = 2
```

Tipos de bifurcaciones I

MUCHO OJO CON LOS TABULADOS EN PYTHON!!!!!!!!!!!!!

- Una rama →

```
if count % 2 == 0:
    print(f'Count is even')
count += 1
```

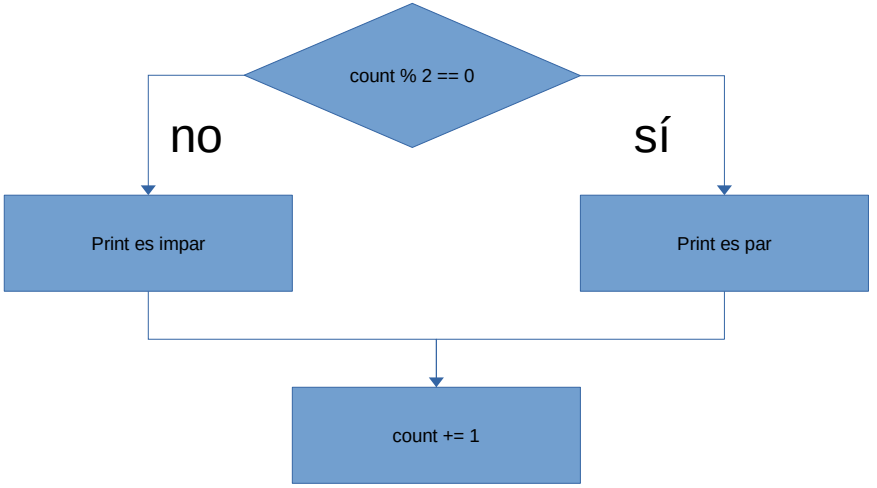


Tipos de bifurcaciones II

MUCHO OJO CON LOS TABULADOS EN PYTHON!!!!!!!!!!!!!

- Dos ramas →

```
if count % 2 == 0:
    print(f'Count is even')
else:
    print(f'Count is odd')
count += 1
```

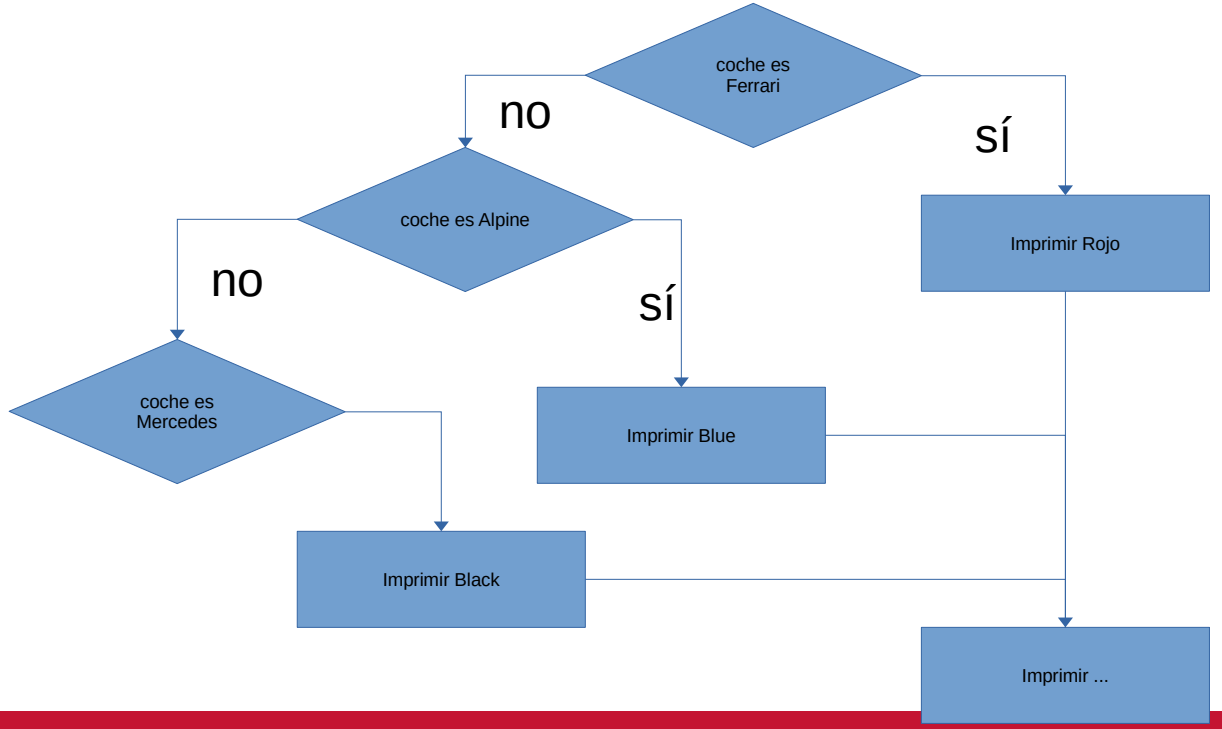


Tipos de bifurcaciones III

MUCHO OJO CON LOS TABULADOS EN PYTHON!!!!!!!!!!!!!

- Multiples ramas →

```
if coche == 'Ferrari':
    print('Red')
elif coche == 'Alpine':
    print('Blue')
elif coche == 'Mercedes':
    print('Black')
elif coche == 'Haas':
    print('White')
else:
    print(f'Unknown')
print('...')
```



Evaluación de las condiciones

- La evaluación de las condiciones es en orden.
  - Es una mala implementación si una condición es subgrupo de otra anterior
  - Es una implementación errónea si una condición es siempre inaccesible

```
if count > 0:
    print("Count is bigger than 0")
elif count > 5:
    print("Count is bigger than 5")
```

```
if is_valid and count > 0:
    print("Is valid and count bigger than 0")
elif is_valid:
    print("Is valid")
```

```
if is_valid:
    print("Is valid")
    if count > 0:
        print("and count bigger than 0")
```

Anidamiento de bifurcaciones

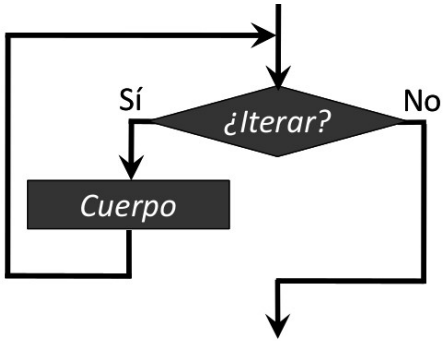
Al margen de la opción de usar un if con múltiples elif, se pueden anidar las bifurcaciones

Ejemplo:

```
if count % 2 == 0:
    print("is even")
    if count > 10:
        print("count is bigger than 10")
    else:
        print("count is smaller than 10")
else:
    print("is odd")
```

Bucles

Conocemos de los diagramas de flujo los **bucles** (repeticiones)



Hay dos tipos: **for** y **while**

Bucle while

Sintaxis

- while condición:

    interior del while...

- **Obligatorio** actualizar la condición en el interior del while. Si la condición es siempre la misma y entra en el bucle... No saldrá (bucle infinito)

```
count = 10
while count >= 0:
    print(count)
    count -= 1
```

Bucle for

Sintaxis

- for element in list_elements:

  interior del while...

- Itera uno por uno la lista de elementos que se le pasa. No se evalúa ninguna condición ni es necesario actualizar ningún valor que afecte a la condición como en el bucle while

```
for number in range(10):
    print(number)
```

```
for character in "This is a text":
    print(character)
```

```
champions = ["Verstappen", "Hamilton", "Vettel", "Alonso"]
for champion in champions:
    print(champion)
```

Funciones

Son un bloque de código aislado que lleva a cabo una actividad

Recibe como parámetros unos datos y devuelve unos datos

- ¿Puede no recibir nada y/o no devolver nada?

Sirven para definir la solución a una acción repetitiva, de forma que:

- El código es más legible
- Solamente hay una base de código para esa acción (imagina que hacemos lo mismo en varios sitios y hacemos una modificación…)

Funciones II

Elementos de una función:

- Nombre del símbolo (nombre de la función, vamos)
- Parámetros de entrada [0..inf)
- Resultado
- Código

Sintaxis:

```
def nombre_de_funcion(parametro1, parametro2, etc...):
    …
```

Funciones III

```
def nombre_de_funcion(parametro1, parametro2, etc...):
    …
```

En Python3 sí puedes indicar el tipo, aunque será solo de caracter informativo. Es decir, el intérprete no analizará la información de tipos que le demos. Nosotros vamos a dar tipos siempre que podamos a las declaraciones de nuestras funciones. Por ejemplo:

```
def nombre_de_funcion(parametro1: int) → bool:
    …
```

Funciones IV

¿Cómo se invoca una función?

Invocar una función es la operación que permite ejecutar el código definido en una función.

Una vez finaliza esa función, el programa continua en la siguiente operación del programa llamante.

Normalemente querremos almacenar el resultado de la ejecución de unafunción

```
dias_de_lluvia = contar_dias_de_lluvia(“Madrid”, 2021)
```

Funciones V – Parámetros de una función

Son los datos que dispone inicialmente la función en su ámbito de ejecución.

En el ámbito de una función: los datos que se reciben se llaman parámetros y son nuevas variables en nuestra función.

Cuando se está invocando a una función: los datos que se entregan se consideran argumentos y son variables en uso cuyos datos queremos entregar a la función.

Debido a la mutabilidad conviene entender la diferencia de perspectiva entre parámetros y argumentos

Parámetros posicionales vs nominales

Una función tiene un número conocido de parámetros. Al invocar una función cada argumento de entrada se corresponde con el parámetro de la función que se encuentre en la misma posición:

```
def any_function(parameter1, parameter2, parameter3, parameter4):
    print(f"parameter1 -> {parameter1}")
    print(f"parameter2 -> {parameter2}")
    print(f"parameter3 -> {parameter3}")
    print(f"parameter4 -> {parameter4}")


any_function(1, 2, 3, 4)
any_function(4, 3, 2, 1)
```

```
/home/josnar/Development/Nebrija/cur
parameter1 -> 1
parameter2 -> 2
parameter3 -> 3
parameter4 -> 4
parameter1 -> 4
parameter2 -> 3
parameter3 -> 2
parameter4 -> 1

Process finished with exit code 0
```

Parámetros posicionales vs nominales

A veces, para evitar confusiones, podemos invocar la función usando los **parámetros nominales**, de forma que al asignar un argumento no importe el orden si no que indiquemos claramente a qué parámetro nos referimos.

```
def any_function(parameter1, parameter2, parameter3, parameter4):
    print(f"parameter1 -> {parameter1}")
    print(f"parameter2 -> {parameter2}")
    print(f"parameter3 -> {parameter3}")
    print(f"parameter4 -> {parameter4}")


any_function(parameter1=1, parameter2=2, parameter3=3, parameter4=4)
any_function(parameter4=4, parameter3=3, parameter2=2, parameter1=1)
```

```
/home/josnar/Development/Nebrija/curso_pyth
parameter1 -> 1
parameter2 -> 2
parameter3 -> 3
parameter4 -> 4
parameter1 -> 4
parameter2 -> 3
parameter3 -> 2
parameter4 -> 1

Process finished with exit code 0
```

Parámetros con valor por defecto

También podemos asignar un valor por defecto a un parámetro. Los parámetros que tienen valores por detecto tienen que situarse los últimos en la lista de parámetros.

```python
def any_function(parameter1, parameter2, parameter3=10, parameter4=11):
    print(f"parameter1 -> {parameter1}")
    print(f"parameter2 -> {parameter2}")
    print(f"parameter3 -> {parameter3}")
    print(f"parameter4 -> {parameter4}")


any_function(1, 2)
any_function(4, 3, 2, 1)
```

```
/home/josnar/Development/Nebrija/curso_pyt
parameter1 -> 1
parameter2 -> 2
parameter3 -> 10
parameter4 -> 11
parameter1 -> 4
parameter2 -> 3
parameter3 -> 2
parameter4 -> 1

Process finished with exit code 0
```

Tipos mutables vs tipos inmutables

Por razones de diseño, en python los tipos básicos son inmutables y los tipos estructurados son los que permiten modificaciones.

Esto significa que una variable de tipo entero definida con un 0, por ejemplo, no puede ser modificada. Y es fácil ver que sí se puede modificar.

¿Cómo es posible?

```
a = 0
print(a)
a += 1
print(a)
```

```
/home/josnar/Development/Nebrija/cu
0
1

Process finished with exit code 0
```

Tipos mutables vs tipos inmutables II

https://docs.python.org/3/library/functions.html#id

id(*object*)

Return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

CPython implementation detail: This is the address of the object in memory.

Raises an auditing event `builtins.id` with argument `id`.

```
a = 0
print(id(a))
print(a)
a += 1
print(id(a))
print(a)
```

```
/home/josnar/Develo
140271796895952
0
140271796895984
1
```

El tipo no muta, sino que se le asigna a la variable un nuevo valor

Tipos mutables vs tipos inmutables III

¿Qué pasa con los tipos estructurados? (mutables según lo que os he dicho)

```python
l = [1, 2, 3]
print(l)
print(id(l))
for element in l:
    print(f"Value is {element} and it id is{id(element)}")

index = 0
while index < len(l):
    l[index] = l[index] * 2
    index += 1

print(l)
print(id(l))
for element in l:
    print(f"Value is {element} and it id is{id(element)}")
```

```
[1, 2, 3]
140709691079424
Value is 1 and it id is140709690360048
Value is 2 and it id is140709690360080
Value is 3 and it id is140709690360112
[2, 4, 6]
140709691079424
Value is 2 and it id is140709690360080
Value is 4 and it id is140709690360144
Value is 6 and it id is140709690360208
```

Paso de parámetros por valor vs referencia

Cuando se pasa como argumento una variable con tipos inmutable, el intérprete copia el valor en la nueva variable que se usará en el ámbito de la función. Esto implica que cualquier cambio que se haga dentro de la función no afectará a la variable original

```
def whatever(anything):
    print(f"The anything inside the function is {anything} with id {id(anything)}")
    anything += 1
    print(f"The anything after adding 1 is {anything} with id {id(anything)}")

anything = 0
print(f"The anything outside the function is {anything} with id {id(anything)}")
whatever(anything)
print(f"Let's check if the anything variable was modified inside the función -> {anything} with id {id(anything)}")
```

```
The anything outside the function is 0 with id 140391355826384
The anything inside the function is 0 with id 140391355826384
The anything after adding 1 is 1 with id 140391355826416
Let's check if the anything variable was modified inside the función -> 0 with id 140391355826384
```

Paso de parámetros por valor vs referencia II

Cuando se pasa como argumento una variable con tipo mutable, el intérprete copia el valor en la nueva variable que se usará en el ámbito de la función. Pero en este caso, cuando hagamos cualquier modificación, dicho cambio será visible en el exterior.

```
def whatever_list(anything):
    print(f"The anything inside the function is {anything} with id {id(anything)}")
    anything.append(3)
    print(f"The anything after appending 3 is {anything} with id {id(anything)}")

anything = [1, 2]
print(f"The anything outside the function is {anything} with id {id(anything)}")
whatever_list(anything)
print(f"Let's check if the anything variable was modified inside the función -> {anything} with id {id(anything)}")
```

```
The anything outside the function is [1, 2] with id 139898399270656
The anything inside the function is [1, 2] with id 139898399270656
The anything after appending 3 is [1, 2, 3] with id 139898399270656
Let's check if the anything variable was modified inside the función -> [1, 2, 3] with id 139898399270656
```