Programación 1º iMAT

Grado en Ingeniería Matemática e Inteligencia Artificial - Comillas ICAI

por David Contreras Bárcena

Tema 9. Funciones

9.1 Introducción

Una función es un bloque de código independiente y encapsulado bajo un nombre único, que recibe un conjunto de parámetros o argumentos de entrada y puede devolver un resultado. Los parámetros o argumentos son los datos de entrada que utilizará la función para ejecutar el conjunto de instrucciones que posee y determinar la salida.

Como hemos visto, Python ofrece una serie de funciones integradas en el lenguaje: int(), bin(), print(), type(), etc. que realizan una funcionalidad muy concreta, reciben un resultado o no, y podemos invocar/llamar repetidas veces.

Por ejemplo, la función **print()** recibe unos argumentos de entrada que determinan la funcionalidad de la función y no devuelve ningún resultado.

```
print("Edad:", 23)
```

```
Edad: 23
```

Por otro lado, las funciones que hemos utilizado mucho como **int()** o **bin()**, recibe un solo argumento y devuelven un resultado, el cual almacenábamos en una variable:

```
numero_binario = bin(5)
numero = int("25")
```

Las funciones son uno de los componentes más importante del paradigma de programación estructurada, ya que ofrecen dos grandes ventajas:

- **Modularización**: permite dividir un programa complejo en una serie de funciones (fragmentos de código) más simples.
- **Reutilización**: cada uno de estas funciones puede ser reutilizado sin necesidad de repetir su código durante el programa.

Al incio de curso se mencionó que las funciones eran los verbos del idioma y las variables los sustantivos. En esa línea, las funciones deberán tener nombres de acciones o verbos: pintar, leer, transformar, etc.

9.2 Estructura de una función

Ahora es el momento de que veamos cómo crear nuestras propias funciones. Los elementos que forman parte de una función son:

- **Nombre**: el nombre de la función debe identificar de forma unívoca la funcionalidad de ese fragmento de código. Será la forma de llamarla durante el programa.
- **Parámetros o argumentos**: es la lista de valores de entrada que puede recibir una función. De una forma más precisa, se refiere al nombre de **parámetro** como a las variables definidas en la función, mientras que los **argumentos** son los valores que recibe la función. En el curso, no daremos mucha importancia a esta distinción ya que dependerá desde donde se vea la función.
- **Instrucciones**: es el bloque de instrucciones Python que realizan la funcionalidad deseada.
- **Retorno**: es la resultado que devuelve un función. Finaliza la ejecución de la función y puede devolver un valor de resultado o no. Cuando una función no devuelve ningún resultado y lo intentamos asignar a una variable, éste tendrá el valor **None**.
- **Docstring**: son los comentarios asociados a una función siguiendo una estructura determinada que nos permiten describir su funcionalidad y estructura (parámetros y resultado).

La palabra reservada de Python para definir funciones es **def**.

```
def nombre (parametro_1:tipo, parametro_2:tipo)->tipo_resultado:
    """Descripción de la funcionalidad de la función

    Args:
        parametro_1 (type): descripción
        parametro_1 (type): descripción

    Returns:
        resultado: descripción

    Raises:
        excepcion: descripción de los errores que puede lanzar

    Examples:
        Ejemplo de ejecución si fuese necesario
    """
    instrucción 1
    instrucción 2
    ....
    instrucción n
    return resultado
```

Se puede omitir cualquiera de los apartados no necesarios del **Docstring**.

9.2.1 Ejemplo de una función básica

Si disponemos un programa que realiza unos cálculos...

In [12]:
```
import math

cateto1a = 2
cateto1b = 3
hipotenusa1 = math.sqrt((cateto1a**2) + (cateto1b**2))
print(hipotenusa1)

cateto2a = 4
cateto2b = 6
hipotenusa2 = math.sqrt((cateto2a**2) + (cateto2b**2))
print(hipotenusa2)
```

```
3.605551275463989
7.211102550927978
```

...podríamos concentrar el código repetido en una función. Como se puede ver, el código resultante queda más limpio y elegante.

Cuando trabajemos con Notebooks, primero definiremos la función y a continuación, después de finalizar el bloque (como ocurre con el resto de estructuras Python) llamaremos a nuestra función en lo que se denomina como **programa principal**.

In [6]:
```
import math

# Definición de la función

def calcular_hipotenusa(cateto1:int, cateto2:int)->float:
    """Calcula la hipotenusa de un triángulo a partir de sus dos catetos

    Args:
        cateto1 (int)
        cateto2 int)

    Returns:
        hipotenusa
    """
    hipotenusa = math.sqrt((cateto1**2) + (cateto2**2))
    return hipotenusa


# Programa principal

hipotenusa = calcular_hipotenusa(2, 3)
print(hipotenusa)
hipotenusa = calcular_hipotenusa(4, 6)
print(hipotenusa)
```

```
3.605551275463989
7.211102550927978
```

9.2.2 Casos particulares en el retorno de valores

Partiendo de este sencillo ejemplo de dos funciones Antes de este tema ya sabíamos llamar a funciones y ahora definir y llamar a las nuestras propias.

In [10]:
```
# Definición de funciones
```

```
def sumar(a:int, b:int)->int:
    return a + b

def mostrar(suma:int)->None:
    print(f"El resultado de la suma es {suma}")

# Programa principal

suma = sumar(2, 3)
mostrar(suma)
```

```
El resultado de la suma es 5
```

Pero, aunque no suele ser habitual, se pueden circustancias algo extrañas, como por ejemplo:

¿Qué pasa si no almacenamos el resultado de una función en ninguna variable? Nada, simplemente lo perdemos...

In [20]:
```
sumar(2, 3)
print("Continúa el programa...")
```

```
Continúa el programa...
```

Y si por el contrario, ¿queremos recoger el resultado de una función que no devuelve nada? En ese caso nos encontraremos con un tipo de dato nuevo denominado **NoneType** que posee el valor **None**.

In [21]:
```
suma = sumar(2, 3)
resultado = mostrar(suma)
print(type(resultado))
print(resultado)
```

```
El resultado de la suma es 5
<class 'NoneType'>
None
```

En un futuro volveremos a ver el significado real de None y otros usos que podemos darle.

9.3 Beneficio de trabajar con funciones

9.3.1 Modularización

Una de las primeras ventajas de trabajar con funciones es que nos permite ordenar el códgo, mejorando su estilo y estructura. Nos permite llegar a la **Programación estructurada** mediante la modularización.

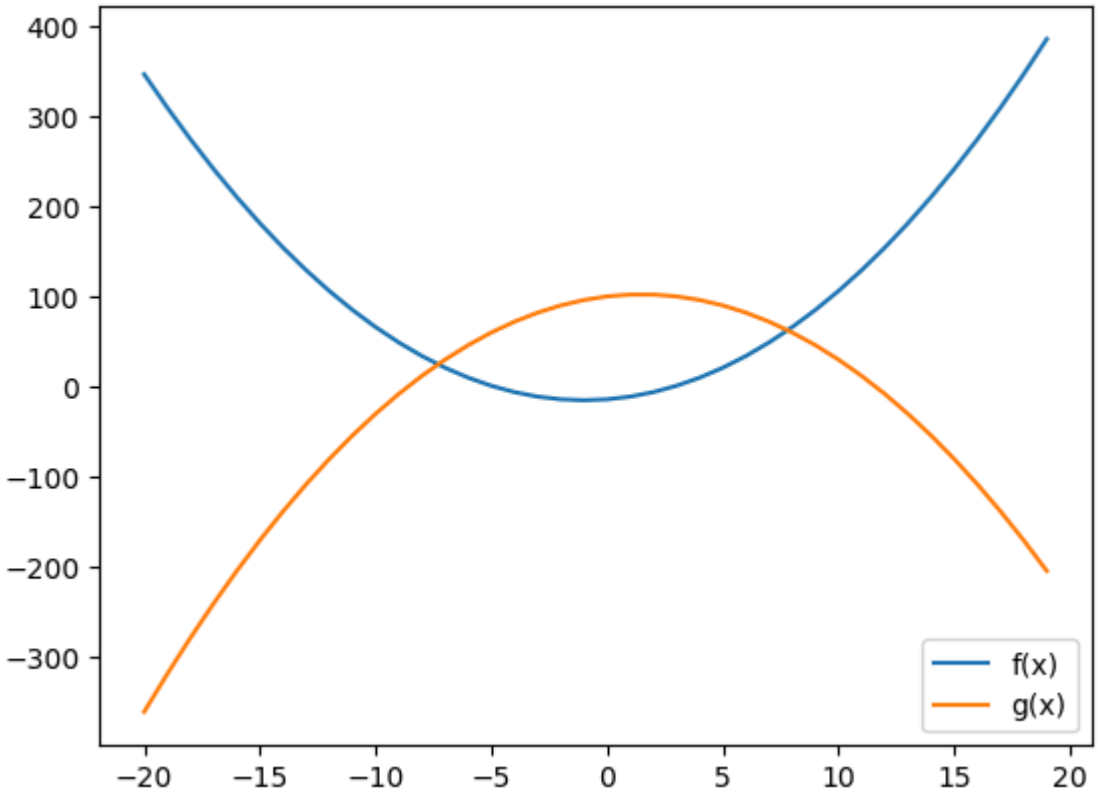
Veamos las ventajas de modularizar o descomponer el código en fragmentos más pequeños, a partir de este ejemplo.

Código de partida

In [26]:
```
import matplotlib.pyplot as plt
```

```python
x1 = -1
while x1 < 1 or x1 > 9:
    x1_str = input("Introduce un número entre el 1 y el 9: ")
    x1 = int(x1_str)
print("Número válido")

x2 = -1
while x2 < 1 or x2 > 9:
    x2_str = input("Introduce un número entre el 1 y el 9: ")
    x2 = int(x2_str)
print("Número válido")

f_x = x1**2 + 2*x1 + 4
g_x = -x2**2 + 3*x2 + 100

print(f"El valor de y de f(x) para x={x1} es {f_x}")
print(f"El valor de y de g(x) para x={x2} es {g_x}")

xs = list(range(-20, 20))
funcion_fx = []
funcion_gx = []
for x in xs:
    funcion_fx.append(x**2 + 2*x - 14)
    funcion_gx.append(-x**2 + 3*x + 100)

plt.plot(xs, funcion_fx, label = "f(x)")
plt.plot(xs, funcion_gx, label = "g(x)")
plt.legend()
plt.show()
```

```
Introduce un número entre el 1 y el 9: 1
Número válido
Introduce un número entre el 1 y el 9: 1
Número válido
El valor de y de f(x) para x=1 es 7
El valor de y de g(x) para x=1 es 102
```



Código modularizado mediante funciones

Únicamente nos vamos a dedicar a dividir el código en funciones, sin optimizar nada más. Solo con este primer paso, obtenemos un programa más legible, con una división de su funcionalidad y perfectamente estructurado en su lógica general, al cual llamaremos **programa principal**.

In [ ]:
```python
import matplotlib.pyplot as plt

# Definición de funciones

def leer_x1()->int:
    """Lee un número por teclado del 1 al 9

    Returns:
        x1 (int): número leido y depurado
    """
    x1 = -1
    while x1 < 1 or x1 > 9:
        x1_str = input("Introduce un número entre el 1 y el 9: ")
        x1 = int(x1_str)
    print("Número válido")
    return x1

def leer_x2()->int:
    """Lee un número por teclado del 1 al 9

    Returns:
        x2 (int): número leido y depurado
    """
    x2 = -1
    while x2 < 1 or x2 > 9:
        x2_str = input("Introduce un número entre el 1 y el 9: ")
```

```
        x2 = int(x2_str)
    print("Número válido")
    return x2

def calcular_valores(x1:int, x2:int)->None:
    """Calcula los valores de la función en esos dos valores de x

    Args:
        x1 (int): primer valor de x
        x2 (int): segundo valor de x
    """
    f_x = x1**2 + 2*x1 + 4
    g_x = -x2**2 + 3*x2 + 100

    print(f"El valor de y de f(x) para x={x1} es {f_x}")
    print(f"El valor de y de g(x) para x={x2} es {g_x}")


def representar_funcion():
    """Genera las listas de valores de los ejes y
    representa las funciones.
    """
    xs = list(range(-20, 20))
    funcion_fx = []
    funcion_gx = []
    for x in xs:
        funcion_fx.append(x**2 + 2*x - 14)
        funcion_gx.append(-x**2 + 3*x + 100)

    plt.plot(xs, funcion_fx, label = "f(x)")
    plt.plot(xs, funcion_gx, label = "g(x)")
    plt.legend()
    plt.show()


# Programa principal

x1 = leer_x1()
x2 = leer_x2()
calcular_valores(x1, x2)
representar_funcion()
```

9.3.2 Reutilización de código

La reutilización de código tiene como objetivo el no repetir código. En el ejemplo anterior se ha llevado el caso al extremo para evitar duplicidad de líneas de código.

🧐 Sobre este ejemplo volveremos para refinarlo aún más.

In [ ]:

```
import matplotlib.pyplot as plt

# Definición de funciones

def leer_valor()->int:
    """Lee un número por teclado del 1 al 9

    Returns:
        x (int): número leido y depurado
```

```
    """
    x = -1
    while x < 1 or x > 9:
        x_str = input("Introduce un número entre el 1 y el 9: ")
        x = int(x_str)
    print("Número válido")
    return x

def calcular_valores(x1:int, x2:int)->None:
    """Calcula los valores de la función en esos dos valores de x

    Args:
        x1 (int): primer valor de x
        x2 (int): segundo valor de x
    """
    f_x = x1**2 + 2*x1 + 4
    g_x = -x2**2 + 3*x2 + 100

    mostrar_salida("f(x)", x1, f_x)
    mostrar_salida("g(x)", x2, g_x)


def mostrar_salida(funcion:str, x:int, y:int)->None:
    """Muestra en pantalla la salida de los valores formateados.

    Args:
        funcion (str): nombre de la función
        x (int): valor de x
        y (int): valor de la función en ese punto x
    """
    print(f"El valor de y de {funcion} para x={x} es {y}")


def representar_funcion()->None:
    """Genera las listas de valores de los ejes y
    representa las funciones.
    """
    xs = list(range(-20, 20))
    funcion_fx = []
    funcion_gx = []
    for x in xs:
        funcion_fx.append(x**2 + 2*x - 14)
        funcion_gx.append(-x**2 + 3*x + 100)

    plt.plot(xs, funcion_fx, label = "f(x)")
    plt.plot(xs, funcion_gx, label = "g(x)")
    plt.legend()
    plt.show()


# Programa principal

x1 = leer_valor()
x2 = leer_valor()
calcular_valores(x1, x2)
representar_funcion()
```

Otro ejemplo de reutilización de código

Partiendo del ejemplo anterior y optimizando el diseño del programa utilizando varios conceptos aprendidos...

In [20]:
```
import math

def calcular_hipotenusa(cateto1:int, cateto2:int)->float:
    """Calcula la hipotenusa de un triángulo a partir de sus dos catetos

    Args:
        cateto1 (int)
        cateto2 (int)

    Returns:
        hipotenusa (float)
    """
    hipotenusa = math.sqrt((cateto1**2) + (cateto2**2))
    return hipotenusa

parejas_catetos = [(2, 3), (4, 6)]
for tupla_catetos in parejas_catetos:
    c1, c2 = tupla_catetos
    hipotenusa = calcular_hipotenusa(c1, c2)
    print(hipotenusa)
```

```
3.605551275463989
7.211102550927978
```

Simplificando el número de líneas del programa...

In [27]:
```
import math

def calcular_hipotenusa(cateto1:int, cateto2:int)->float:
    """Calcula la hipotenusa de un triángulo a partir de sus dos catetos

    Args:
        cateto1 (int)
        cateto2 (int)

    Returns:
        hipotenusa (float)
    """
    hipotenusa = math.sqrt((cateto1**2) + (cateto2**2))
    return hipotenusa

parejas_catetos = [(2, 3), (4, 6)]
for tupla_catetos in parejas_catetos:
    print(calcular_hipotenusa(tupla_catetos)
```

```
3.605551275463989
7.211102550927978
```

9.4 Aspectos particulares de las funciones

9.4.1 Llamada a una función explicitando el nombre de los parámetros

Nunca lo hemos utilizado hasta ahora, pero Python permite una llamada a las funciones de una forma semántica, escribiendo de forma explícita el nombre de los parámetros al pasar sus valores.

```
def funcion1(parametro_1:int, parametro_2:int):
    ....

funcion1(parametro_1 = valor, parametro_2 = valor)
```

Por ejemplo, en una función básica como puede ser la siguiente...

In [23]:
```
def dividir(a:int, b:int)->float:
    return a / b

dividir(4, 2)
```

Out[23]: 2.0

Podemos indicar como se vinculan los parámetros con sus valores.

In [25]:
```
def dividir(a:int, b:int)->float:
    return a / b

dividir(a = 4, b = 2)
```

Out[25]: 2.0

Esta sintáxis puede resultar útil cuando se trata de una función de muchos parámetros y cuando se quiere expresar explícitamente a qué parámetro se asocia cada valor. Esta forma permitirá no tener que respetar el orden de los parámetros.

In [28]:
```
dividir(b = 4, a = 2)
```

Out[28]: 0.5

9.4.2 Valores por defecto en los parámetros

Las funciones pueden tener valores preestablecidos o por defecto para algunos de sus parámetros, de manera que se pueden omitir a la hora de llamar a la función.

En el caso de la función siguiente la funcion1 podrá recibir un parámetro o dos. En el caso de recibir un solo el valro de *parametro_1*, *parametro_2* tomará el valor por defecto asociado.

```
def funcion1 (parametro_1, parametro_2 = valor_por_defecto):
    """DOCSTRING_DE_LA_FUNCION"""
    instrucción 1
    instrucción 2
    ….
    instrucción n
    return expresion
```

Ejemplo

In [30]:
```python
def formatear_nombre(nombre:str, apellido1:str, apellido2:str, formato:int) -> s
    """Devuelve el nombre completo concatendado según el formato especificado

    Args:
        nombre (str): nombre de la persona
        apellido1 (str): primer apellido
        apellido2 (str): segundo apellido
        formato (int): puede tomar dos valores:
                1 -> nombre apellido1 apellido2
                2 -> apelidos1 apellido2, nombre
    Return:
        salida (str): nombre completo según el valor de formato
    """
    salida = ""
    if formato == 1:
        salida = f"{nombre} {apellido1} {apellido2}"
    elif formato == 2:
        salida = f"{apellido1} {apellido2}, {nombre}"
    return salida
```

In [25]:
```python
print(formatear_nombre("Luis", "Fernández", "Sánchez", 1))
print(formatear_nombre("Luis", "Fernández", "Sánchez", 2))
```

```
Luis Fernández Sánchez
Fernández Sánchez, Luis
```

Utilizando este recurso que nos facilita el lenguaje,se entenderá que el formato por defecto es el número 1...

In [40]:
```python
def formatear_nombre(nombre:str, apellido1:str, apellido2:str, formato:int = 1)
    salida = ""
    if formato == 1:
        salida = f"{nombre} {apellido1} {apellido2}"
    elif formato == 2:
        salida = f"{apellido1} {apellido2}, {nombre}"
    return salida
```

In [42]:
```python
print(formatear_nombre("Luis", "Fernández", "Sánchez"))  # Se puede omitir el úl
print(formatear_nombre("Luis", "Fernández", "Sánchez", 2))  # Y cuando se desee
```

```
Luis Fernández Sánchez
Fernández Sánchez, Luis
```

¿Y si quisiéramos hacer dos de los parámetros opcionales con valores por defecto?

In [45]:
```python
def formatear_nombre(nombre:str, apellido1:str, apellido2:str = "", formato:int
    salida = ""
    if formato == 1:
        salida = f"{nombre} {apellido1} {apellido2}"
    elif formato == 2:
        salida = f"{apellido1} {apellido2}, {nombre}"
    return salida
```

In [47]:
```python
print(formatear_nombre("James", "Smith"))
```

```
James Smith
```

¿Y si queremos mostrar un nombre con un solo apellido (James Smith) para que se imprima con el formato 2?

In [50]:
```
print(formatear_nombre("James", "Smith", 2))
```

```
James Smith 2
```

Vemos que en el caso anterior se ha producido un cast/conversión de apellido2 a int tomando el valor 2. Ojo con estas acciones, porque conllevan errores de ejecución complejos de identificar.

❌ ¿Qué pasaría si en lugar de utilizar f-string, utilizáramos el operador de concatenación +, el cual requiere que todos los elementos sean string?

In [53]:
```
def formatear_nombre(nombre:str, apellido1:str, apellido2:str = "", formato:int
    salida = ""
    if formato == 1:
        salida = nombre + " " + apellido1 + " " + apellido2
    elif formato == 2:
        salida = apellido1 + " " + apellido2 + ", " + nombre
    return salida
```

In [55]:
```
print(formatear_nombre("James", "Smith", 2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[55], line 1
----> 1 print(formatear_nombre("James", "Smith", 2))

Cell In[53], line 4, in formatear_nombre(nombre, apellido1, apellido2, formato)
      2 salida = ""
      3 if formato == 1:
----> 4     salida = nombre + " " + apellido1 + " " + apellido2
      5 elif formato == 2:
      6     salida = apellido1 + " " + apellido2 + ", " + nombre

TypeError: can only concatenate str (not "int") to str
```

En el caso anterior, como os podréis imaginar, deberemos utilizar la sintaxis que hemos visto en el apartado anterior.

In [58]:
```
print(formatear_nombre("James", "Smith", formato = 2))
```

```
Smith , James
```

Por lo que se podría llegar a este grado de explicitación de los argumentos, lo cual no es muy práctico, pero Python lo soporta.

In [87]:
```
print(formatear_nombre(nombre = "Luis", apellido1 = "Fernández", apellido2 = "Sá
```

```
Fernández Sánchez, Luis
```

Pudiendo cambiar el orden de los parámetros, como se comentó anteriormente.

In [88]:
```
print(formatear_nombre(apellido2 = "Sánchez", formato = 2, nombre = "Luis", apel
```

```
Fernández Sánchez, Luis
```

9.4.3 Parámetros arbitrarios

Se utiliza esta forma cuando no se sabe el número de argumentos que se van a recibir, pudiendo ir de o a infinitos. En otros lenguajes esta funcionalidad se denomina como **varargs**. En este caso, se definirá un solo parámetro con un **asterisco** que será de tipo **tupla**, recibiendo en esta estructura el conjunto de parámetros recibidos.

```
def nombre (*parametros):
    """DOCSTRING_DE_FUNCION"""
    instrucción 1
    instrucción 2
    ….
    instrucción n
    return expresion
```

In [94]:
```
def concatenar(*palabras):
    salida = ""
    print(f"Solo se muestra como ejemplo: {palabras} - Tipo: {type(palabras)}")
    for palabra in palabras:
        salida += palabra + " "
    return salida
```

In [95]:
```
print(concatenar("a", "b", "c"))
```

```
Solo se muestra como ejemplo: ('a', 'b', 'c') - Tipo: <class 'tuple'>
a b c
```

In [54]:
```
print(concatenar("a", "b", "c", "d", "e", "f"))
```

```
a b c d e f
```

Si en lugar de utilizar un asterisco en los paréntesis, se utilizan dos, en lugar de una lista de posibles parámetros, se podrá pasar un diccionario de nombre de parámetro y valor del mismo.

Como estándar de facto, ee suele llamar a este argumento **kwargs**: keyword arguments.

```
def nombre (**kwargs):
    """DOCSTRING_DE_FUNCION"""
    instrucción 1
    instrucción 2
    ….
    instrucción n
    return expresion
```

In [9]:
```
def concatenar(**palabras):
    salida = ""
    print(f"Solo se muestra como ejemplo: {palabras} - Tipo: {type(palabras)}")
    for k, v in palabras.items():
        if k == "b" or k == "c":
            salida += v + " "
    return salida
```

In [10]:
```
print(concatenar(a = "hola", b = "adiós", c = "hasta luego"))
```

```
Solo se muestra como ejemplo: {'a': 'hola', 'b': 'adiós', 'c': 'hasta luego'} - T
ipo: <class 'dict'>
adiós hasta luego
```

9.4.5 Retorno

Ya se ha visto que la forma común y general a todos los lenguajes de programación es el uso de **return** para devovler el resultado de una función. En Python existe la particularidad de que se pueden devolver varios valores de retorno en forma de una tupla.

Ojo con esta funcionalidad, porque se utilizará en ocasiones muy contadas, cuando realmente el elemento resultante deba ser realmente una tupla, no porque se desee devlver tres valores cualesquiera que no tengan ninguna relación entre sí, por ejemplo.

```
def nombre (parametro_1:tipo, parametro_2:tipo, ...) -> tipo:
    """DOCSTRING_DE_FUNCION"""
    instrucción 1
    instrucción 2
    ……
    instrucción n
    return resultado_1, resultado_2, ...
```

A continuación se ve un ejemplo de función que devuelve más de un resultado.

In [74]:
```
def crear_punto(x:int, y:int)->tuple[int]:
    if x < 0 or x > 10:
        x = 5
    if y < 0 or y > 10:
        y = 5
    return x, y
```

In [76]:
```
punto = crear_punto(4, 6)
print(type(punto))
print(punto)
```

```
<class 'tuple'>
(4, 6)
```

Como se vio en temas anteriores, se pueden recibir como múltiples variables.

In [111…
```
x, y = crear_punto(-1, 11)
print(x, y)
```

```
5 5
```

Que podría ser equivalente a crear la tupla y devolverla como un elemento único.

In [96]:
```
def crear_punto(x:int, y:int) -> tuple[int, int]:
    if x < 0 or x > 10:
        x = 5
    if y < 0 or y > 10:
        y = 5
    punto = (x, y)
    return punto
```

```
x, y = crear_punto(4,7)
print(x, y)
```

```
4 7
```

El ejemplo anterior nos hace pensar que realmente se puede devolver cualquier tipo de objeto como resultado de una función.

In [77]:
```
def dame_muchos_numeros(numero=10):
    numeros = []
    for i in range(1, numero+1):
        numeros.append(i)
    return numeros
```

In [78]:
```
print(dame_muchos_numeros())
```

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

9.5 Estructura de un programa

Ahora que ya conocemos que un programa se puede modularizar en en muchas funciones, veamos como diseñar correctamente un programa.

9.5.1 Estructura de un programa en un Notebook

Ya hemos visto que la secuencialidad de las funciones y programa principal es la siguiente. El kernel de Jupyter ejecuta la secuencia de código que no pertenece a ninguna función.

```
def f1():
    instrucciones

def f2():
    instrucciones

def f3():
    instrucciones

def f4():
    instrucciones

# Programa principal
instrucciones
f1()
instrucciones
f2()
instrucciones
f3()
instrucciones
f4()
```

In [98]:
```
def multiplicar(a:int, b:int) -> int:
    return a * b

def sumar(a:int, b:int) -> int:
```

```
    return a + b

# Programa principal
a = 4
b = 2
multiplicacion = multiplicar(a, b)
suma = sumar(a, b)
print(f"{a} + {b} = {suma}")
print(f"{a} * {b} = {multiplicacion}")
```

```
4 + 2 = 6
4 * 2 = 8
```

9.5.2 De Notebooks a ficheros Python

Los notebooks son un gran elemento de escritura de código rápido para el desarrollo de prototipos, pequeños programas, aprendizaje de un lenguaje, etc. pero no es la mejor herramienta para el desarrollo de programas de software.

Para desarrollar programas medianos a grandes utilizaremos los programa llamados como Entornos Integrados de Desarrollo (IDE). En nuestro caso, utilizaremos Visual Studio Code.

La unidad de programación de los Notebooks son los ficheros con extensión .ipynb, mientras que en los IDE (en cualquiera) serán los archivos .py.

Un programa de software se escribirá siempre en un conjunto de ficheros .py para mejorar su legibilidad y modularidad. Así, el programa principal se definirá en un fichero .py que se podrá llamar de forma general como main.py o app.py o con la finalidad del programa: juego.py, factorial.py, calculadora.py, ...). Las funciones complementarias se escribirán en otros ficheros .py, a los cuales denominaremos **módulos** (funciones.py, calculos.py, validacion.py, conversiones.py, etc.)

Esto siginificará que pasaremos de escribir nuestros programas de un solo fichero .ipynb a varios .py.

Cuando poseemos distintos ficheros .py deberemos indicar cuál de ellos es el programa pricipal o el punto de inicio de ejecución de nuestros programas. Esta indicación la haremos mediante el código que marca su punto de inicio, la estructura condicional que indica el bloque de inicio **main**:

```
if __name__ == "__main__":
    codigo_del_programa_principal
```

9.6 Ámbito de las funciones

Al trabajar en un notebook o en un mismo fichero .py sin funciones, el ámbito de actuación de las variables es global a todo el programa, es decir, pueden ser usadas en todo el programa.

En el momento que utilizamos funciones, el ámbito de las variables definidas en ellas, tendrán un ámbito local, es decir, solo existirán dentro de la función.

In [1]:
```
def funcion():
    variable_local = 25
    print(variable_local)

if __name__ == "__main__":
    funcion()
    print(variable_local)  # Esta variable no existe fuera del ámbito de la func
```

```
25
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_6208\6299227.py in <module>
      5 if __name__ == "__main__":
      6     funcion()
----> 7     print(variable_local)  # Esta variable no existe fuera del ámbito de
la función

NameError: name 'variable_local' is not defined
```

Cuando se define una variable fuera de una función ésta tendrá un ámbito global a todo el programa:

In [6]:
```
variable_global = 10

def funcion():
    variable_local = 25
    print("Local:", variable_local)
    print("Global en función:", variable_global)

if __name__ == "__main__":
    funcion()
    print("Global en programa ppal:", variable_global)
```

```
Local: 25
Global en función: 10
Global en programa ppal: 10
```

Como las variables creadas en una función tienen un ámbito local, existen solo en la función, aunque se cree una variable con el mismo nombre que una variable global será tratada siempre como una local.

In [2]:
```
variable = 10  # Variable global

def funcion():
    variable = 25  # Variable local con el mismo nombre que la global
    print("Variable en función:", variable)

if __name__ == "__main__":
    funcion()
    print("Variable en programa ppal:", variable)
```

```
Variable en función: 25
Variable en programa ppal: 10
```

¿Y si queremos modificar el valores de la variable global dentro de la función? En ese caso utilizaremos el modificar **global** sobre la variable en la función, para que el compilador no se genere una nueva variable local y acceda a la global.

In [10]:
```python
variable = 10  # Variable global

def funcion():
    global variable
    variable = 25  # Variable global
    print("Variable en función:", variable)

if __name__ == "__main__":
    funcion()
    print("Variable en programa ppal:", variable)
```

```
Variable en función: 25
Variable en programa ppal: 25
```

9.6.1 Referencia de objetos en memoria

Pero ¿cómo se almacenan las variables en memoria? Cada vez que se define una variable se realiza una reserva en memoria para almacenar su valor, pero no siempre igual en función del tipo de dato de la variable. De una forma general, se podrá dividir el comportamiento de los objetos/variables que representan tipos de datos básicos (int, float, bool, str, ...), del resto (list, tuple, dict, ...).

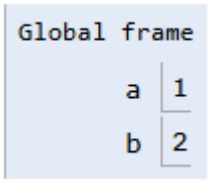
Sin funciones

Para entender esta gestión de memoria, primero trabajaremos en el contexto del programa principal (memoria global), sin funciones. Aquí, cada variable definida que almacene un tipo de dato básico tendrá su espacio de memoria para almacenar su valor. La asignación de un nuevo valor, significará una copia de la variable, como era de esperar. Se puede observar en el siguiente ejemplo al crear una nueva variable `b` con el valor de `a` como se copia su valor.

In [119…
```python
a = 1
b = a
b += 1
print(f"Variable a: {a} - Variable b: {b}")
```

```
Variable a: 1 - Variable b: 2
```

La representación en memoria será la siguiente:



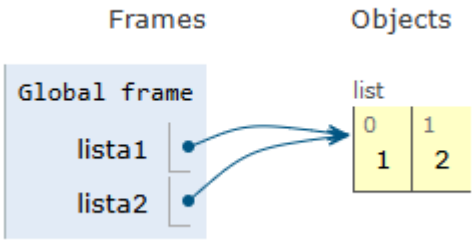
Cuando trabajamos con los objetos complejos, como las listas, nos encontramos con que las asignaciones no producen copias del siguiente, si no que una vinculación o referencia sobre la misma.

`lista2` apuntará a la misma dirección de memoria que `lista1`, por lo que un cambio en una, producirá un cambio en la otra.

In [132…

```
lista1 = [1]
lista2 = lista1
lista1.append(2)
print(f"List1: {lista1} - List2: {lista2}")
```

```
List1: [1, 2] - List2: [1, 2]
```



Con funciones

Cada vez que enviamos un argumento a una función se produce una asignación de una variable a otra, por lo que ocurreirá lo mismo que hemos visto antes. De esta forma, se dice que el envío de los argumentos serán:

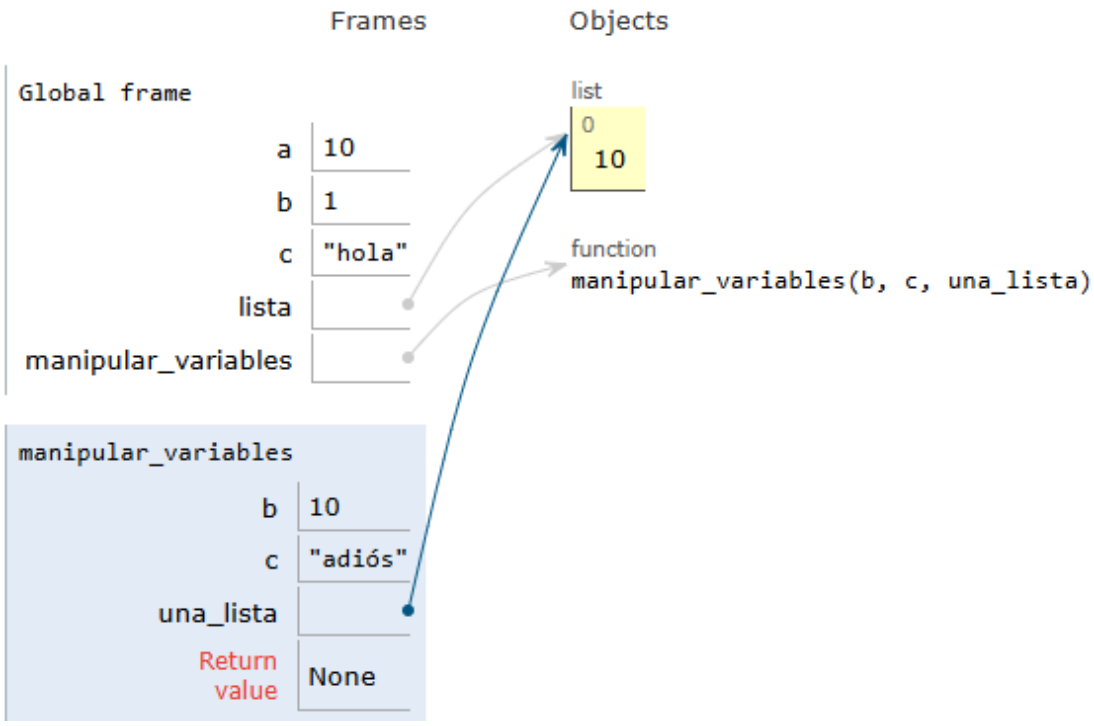
- Tipos básicos: por valor (generación de copias). El parámetro de la función se comportará como una variable local, a no ser que se especifique lo contrario con global.
- Tipos complejos: por referencia (las variables apuntarán a la misma dirección de memoria). El parámetro de la función se comportará siempre como una variable global.

Vease con el siguiente ejemplo:

In [160…

```
a = 1
b = 1
c = "hola"
lista = []

def manipular_variables(b:int, c:str, una_lista:list) -> None:
    global a
    a = 10    # Misma variable global
    b = 10    # Copia local de la variable
    c = "adiós"    # Copia local de la variable
    una_lista.append(10)

manipular_variables(b, c, lista)
print(a, b, c, lista)
```

```
10 1 hola [10]
```



9.7 Módulos

9.7.1 Creación básica

Un módulo es un fichero .py que contienen la definición de funciones Python. Hay que tener en cuenta que, en ocasiones, las funciones que definamos en estos módulos, podrán ser llamadas por más de un programa principal.

Definiremos el ejemplo anterior en una estructura de módulos, creando los siguientes archivos .py.

En primer lugar se crean las funciones en su fichero correspondiente.

Fichero calculos.py

```
def multiplicar(a, b):
    return a * b

def sumar(a, b):
    return a + b
```

A continuación, en el fichero que define el programa principal y que llamará a las funciones creadas, deberemos indicarle el nombre del fichero donde se encuentran estas funciones. Esta acción la realizaremos mediante la paralabra reservada **import** y el nombre del fichero sin extensión: **calculos**.

Como os podréis imaginar si omitís la sentencia import, el compilador de Python no sabrá donde encontrar las funciones definidas.

Por último, teniendo en cuenta que un programa principal podrá llamar a distintas funciones de diferentes módulos, estaremos obligados a indicar al compilador en qué módulo se encuentra cada función escribiendo: **modulo.funcion()**.

Fichero app.py

```
import calculos

if __name__ == "__main__":
    a = 4
    b = 2
    multiplicacion = calculos.multiplicar(a, b)
    suma = calculos.sumar(a, b)
    print(f"{a} + {b} = {suma}")
    print(f"{a} * {b} = {multiplicacion}")
```

9.7.2 Otras formas de importar módulos

Si el nombre del módulo es muy largo, se puede crear un alias o abreviatura con la instrucción **as**, como por ejemplo:

```
import calculos as calc

if __name__ == "__main__":
    a = 4
    b = 2
    multiplicacion = calc.multiplicar(a, b)
    suma = calc.sumar(a, b)
    print(f"{a} + {b} = {suma}")
    print(f"{a} * {b} = {multiplicacion}")
```

Si solo vamos a utilizar un módulo y tenemos perfectamente identificadas sus funciones, podemos trabajar directamente con ellas. Esta forma de trabajar no se recomienda cuando trabajemos con varios módulos.

```
from calculos import *

if __name__ == "__main__":
    a = 4
    b = 2
    multiplicacion = multiplicar(a, b)
    suma = sumar(a, b)
    print(f"{a} + {b} = {suma}")
    print(f"{a} * {b} = {multiplicacion}")
```

Cuando los módulo se encuentran dentro de subdirectorio o subcarpetas, se dice que se encuentran almacenados y estructurados en **paquetes**.

```
import nombre_paquete.nombre_modulo
```

Ahora supongamos que guardamos el módulo **calculos.py** dentro del subdirectorio (paquete) *funciones*.

```
import funciones.calculos as calc

if __name__ == "__main__":
    a = 4
    b = 2
    multiplicacion = calc.multiplicar(a, b)
    suma = calc.sumar(a, b)
```

```
print(f"{a} + {b} = {suma}")
print(f"{a} * {b} = {multiplicacion}")
```

9.7.3 Iimportar módulos en paquetes

Recordemos como en los ejemplo anteriores, hemos llamado a módulos en algunos ejemplos:

Por ejemplo, cuando escribíamos la siguiente sentencia...

```
import matplotlib.pyplot as plt
```

...realmente estábamos indicando que en el módulo **pyplot.py** se encontraba en el **paquete** del compilador de Python **matplotlib** y lo queríamos referenciar de forma abreviada como **plt**.

Las funciones show() o legend() se encuentran dentro del módulo **pyplot** (fichero pyplot.py).

```
plt.legend()
plt.show()
```

9.7.4 Importar variables de módulos

Si se desea acceder a variables globales definidas en otros ficheros .py, se deberá trabajar de forma similar a la forma de importar funciones en módulos.

Fichero funciones.py

```
variable_externa = 50
```

Fichero app.py

```
from funciones import variable_externa
print(variable_externa)
from funciones import *
print(variable_externa_1)
print(variable_externa_2)
funcion_externa()
```

In [ ]: