Tema 6. Funciones

November 19, 2020

1 Introducción

- Definición: un pequeño programa que puedo usar en otro programa
- Ejemplos de funciones: `type()`, `len()`, `print()`, `id()`, `max()`, `min()`, `input()`, `int()`
- Otros nombres: procedimientos, sub-rutinas, sub-programas
- Su objetivo principal es organizar mejor el código
- Permiten: reutilización del código, descomposición, ocultación de la implementación

2 Reutilización de código

- Las funciones me permiten usar el mismo código en distintas partes del programa (cuando un bucle no es la mejor solución)
- Una función está compuesta por tres partes: cabecera, documentación (opcional), cuerpo

```
def nombre():
    """Explicación de qué hace la función"""      instruccion1
    instruccion2
     ...
    instruccionN
```

- Cuando declaro una función, no hace nada, solamente guarda ese código en memoria
- Una vez declarada, la puedo utilizar
- Es obligatorio declarar las funciones antes de usarlas (en otros lenguajes no lo es). La regla de estilo dice que deberían ir todas juntas al principio del programa
- Reglas de estilo: nombres en minúsculas y si son más de una palabra uso guion bajo: `esto_es_un_nombre_de_funcion_correcto` (se podría usar también `estoEsUnNombreDeFuncionAlgoMenosCorrecto`)

[1]:
```
# Ejemplo de un programa que pide a gritos usar funciones
print('Hace un día maravilloso en Cabo Cañaveral')
# Desde aquí hasta la línea 8, se repiten luego
print("Todo listo para el lanzamiento")
print('Comienza la cuenta atrás')
for i in range(10, -1, -1):
    print(i, end=" ")
print('¡Despegamos!')
print("¡Houston, Houston, tenemos un problema!")
print("Abortamos la misión, mañana probaremos otra vez")
```

```
print('Hoy vuelve a hacer un día maravilloso en Cabo Cañaveral')
print("Todo listo para el lanzamiento")
print('Comienza la cuenta atrás')
for i in range(10, -1, -1):
    print(i, end=" ")
print('¡Despegamos!')
print("¡Houston, todo bien por aquí!")
print("Mira la luna, parece un queso")
```

```
Hace un día maravilloso en Cabo Cañaveral
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
Hoy vuelve a hacer un día maravilloso en Cabo Cañaveral
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, todo bien por aquí!
Mira la luna, parece un queso
```

[1]:
```
# Cambiamos el programa anterior para usar funciones
def lanzamiento():
    """Esta función simula el lanzamiento de un cohete"""
    print("Todo listo para el lanzamiento")
    print('Comienza la cuenta atrás')
    for i in range(10, -1, -1):
        print(i, end=" ")
    print('¡Despegamos!')
# Si ejecuto el programa hasta aquí, no hace nada. Lo único que
# Python hace cuando encuentra la definición de una función es
# guardarla en memoria, pero no la ejecuta

print('Hace un día maravilloso en Cabo Cañaveral')
# Invoco la función, Python ejecuta el código
lanzamiento()
print("¡Houston, Houston, tenemos un problema!")
print("Abortamos la misión, mañana probaremos otra vez")

print('Hoy vuelve a hacer un día maravilloso en Cabo Cañaveral')
# La vuelvo a invocar
lanzamiento()
print("¡Houston, todo bien por aquí!")
print("Mira la luna, parece un queso")
```

```
Hace un día maravilloso en Cabo Cañaveral
```

```
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
Hoy vuelve a hacer un día maravilloso en Cabo Cañaveral
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, todo bien por aquí!
Mira la luna, parece un queso
```

3 Parámetros

- Una función puede tener parámetros. Permiten que la función haga cosas ligeramente diferentes cada vez que la invoco
- `def nombre_funcion(parametro1, parametro2, parametro3):`
- Los parámetros se colocan en la cabecera y son como variables que no hace falta inicializar porque se les dará valor obligatoriamente cuando se invoque la función
- Las reglas de nombres de los parámetros son las mismas que las de las variables
- ¿Cuál es el valor inicial de un parámetro? El que se le da cuando se invoca la función, hasta ese momento no tiene ninguno porque no se necesita todavía
- Cuando invoco la función tengo que dar valor a los parámetros. Es obligatorio dar valor a todos ellos, ni a más ni a menos

[3]:
```python
# Ejemplo de función con 1 párametro
def lanzamiento(inicio):
    """Esta función simula el lanzamiento de un cohete"""
    print("Todo listo para el lanzamiento")
    print('Comienza la cuenta atrás')
    for i in range(inicio, -1, -1):
        print(i, end=" ")
    print('¡Despegamos!')
# Si ejecuto el programa hasta aquí, no hace nada. Lo único que
# Python hace cuando encuentra la definición de una función es
# guardarla en memoria, pero no la ejecuta

print('Hace un día maravilloso en Cabo Cañaveral')
# Invoco la función, Python ejecuta el código. Como tiene un
# parámetro, estoy obligado a dar valor valor al parámetro para
# poder invocarla. Si no le doy valor, salta un error
lanzamiento(inicio = 10)
print("¡Houston, Houston, tenemos un problema!")
print("Abortamos la misión, mañana probaremos otra vez")

print('Hoy vuelve a hacer un día maravilloso en Cabo Cañaveral')
# La vuelvo a invocar. Puedo poner simplemente el valor, sin el
```

```
# nombre. Esta es la forma más habitual
lanzamiento(20)
print("¡Houston, todo bien por aquí!")
print("Mira la luna, parece un queso")
```

```
Hace un día maravilloso en Cabo Cañaveral
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
Hoy vuelve a hacer un día maravilloso en Cabo Cañaveral
Todo listo para el lanzamiento
Comienza la cuenta atrás
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, todo bien por aquí!
Mira la luna, parece un queso
```

[ ]:
```
# Ejemplo de función con 2 párametros
def lanzamiento(inicio, exito):
    """Esta función simula el lanzamiento de un cohete"""
    print("Todo listo para el lanzamiento")
    print('Comienza la cuenta atrás')
    for i in range(inicio, -1, -1):
        print(i, end=" ")
    print('¡Despegamos!')
    if exito:
        print("¡Houston, todo bien por aquí!")
        print("Mira la luna, parece un queso")
    else:
        print("¡Houston, Houston, tenemos un problema!")
        print("Abortamos la misión, mañana probaremos otra vez")
# Si ejecuto el programa hasta aquí, no hace nada. Lo único que
# Python hace cuando encuentra la definición

print('Hace un día maravilloso en Cabo Cañaveral')
# Primera forma de invocar la función, pongo el nombre del parámetro
# y su valor
lanzamiento(inicio = 10, exito = False)
print('Hoy vuelve a hacer un día maravilloso en Cabo Cañaveral')
# Segunda forma, sin poner los nombres. Asigna los valores a los
# parámetros en orden: el primer valor al primer parámetro, etc.
# Siempre el número de valores tiene que ser igual al de parámetros
lanzamiento(20, True)
```

4 Especificando los tipos de los parámetros

- Cuando voy a usar una función ¿cómo sé el tipo de cada parámetro?
- Tres formas de especificar el tipo de los parámetros
    - Añadiéndolo en la documentación de la función `@param nombre_parametro: tipo` Es solamente indicativo, se podrían pasar parámetros de otro tipo
    - Usando anotaciones en la cabecera de la función `parametro: tipo`. En otros lenguajes son obligatorias. En Python es solamente indicativo, se podrían pasar parámetros de otro tipo
    - Controlando realmente el tipo mediante la función `type()` o `isinstance()`
    - Este curso deberiais usar siempre las dos primeras

[18]:
```
# Primera y segunda forma de especificar el tipo de los parámetros
# no son obligatorias para el que llama la función
def maximo(primero: float, segundo: float):
    """ Esta función imprime el máximo de 2 números
    @param primero: un entero o flotante
    @param segundo: un entero o flotante"""
    if primero > segundo:
        print(primero)
    else:
        print(segundo)

# Tercera forma, compruebo efectivamente que los tipos son los
# correctos
def maximo(primero: float, segundo: float):
    """ Esta función imprime el máximo de 2 números
    @param primero: un entero o flotante
    @param segundo: un entero o flotante
    Si los tipos no son correctos no hace nada"""
    if ((type(primero) == int or type(primero) == float)
            and (type(segundo) == int or type(segundo)== float)):
        if primero > segundo:
            print(primero)
        else:
            print(segundo)

maximo(1, 1.5)
# Esto no hace nada
maximo(3, "hola")
```

```
1.5
```

5 Funciones que devuelven cosas (return)

- Lo útil en una función no es que imprima el resultado sino que lo devuelva. Las funciones en general no imprimen, siempre devuelven el valor para que yo haga lo que quiera con él

- Para ello utilizo la instrucción de ramificación `return`
- **Devolver** no es **imprimir**
- En general, salvo que haya una buena razón las funciones devuelven algo, no lo imprimen
- Si invoco una función que tiene un return, debería asignar esa función a una variable para guardar el valor que me devuelve o imprimirlo
- Es conveniente utilizar anotaciones para decir el tipo que devuelve la función (`-> tipo:`) y también ponerlo en la documentación (`@return`)
- Puedo devolver más de una cosa: usando comas para separar. Lo que se devuelve es una tupla
- Importante: una vez se encuentra un return la función termina y devuelve el valor, nada de la función se ejecuta después

[15]:
```python
# Ejemplo de una función que devuelve el máximo de 2 números
def maximo(primero: float, segundo: float)-> float:
    """ Esta función devuelve el máximo de 2 números
    @param primero: un entero o flotante
    @param segundo: un entero o flotante
    @return el máximo de los dos números (float)
    Si los tipos no son correctos no hace nada"""
    if ((type(primero) == int or type(primero) == float)
            and (type(segundo) == int or type(segundo)== float)):
        if primero > segundo:
            # Devuelvo el valor de primero
            return primero
        else:
            # Devuelvo el valor de segundo
            return segundo
# Como devuelvo el valor en lugar de imprimirlo, ahora puedo usarla
# para calcular el máximo de tres números
resultado = maximo(2, 4)
resultado2 = maximo(resultado, 3)
print(resultado2)
# También puedo directamente imprimir el resultado sin guardarlo en
# una variable
print(maximo(5,6))
```

```
4
6
```

[19]:
```python
# Función que devuelve dos valores
def primero_y_ultimo(lista: list)-> tuple:
    """Esta función devuelve el primer y último elemento de
    una lista"""
    # Compruebo que es una lista
    if type(lista) != list:
        # Si no lo es, termino la función y devuelvo None
        return None
    primero = lista[0]
```

```
    ultimo = lista[-1]
    # Cuando devuelvo varios valores, lo que devuelvo es una tupla
    # Podría devolver una lista si lo pusiera entre []
    return primero, ultimo
    # Esto no se ejecuta, porque cuando encuentra un return la
    # función termina
    # A esto se le llama código muerto y es un error muy común
    # Los lenguajes compilados lo detectan automáticamente y no
    # ejecutan
    print("La función ha terminado")

a = primero_y_ultimo([1, 2, 4, 5, 67])
print("El tipo del dato que devuelve la función es", type(a))
print("El primer y último elemento son", a)


# Función que termina antes si encuentra lo que busca
def esta(lista: list, elemento: int)-> bool:
    """Me dice si un elemento está en una lista. Es una
    implementación alternativa del operador in"""
    for el in lista:
        # Si lo encuentra devuelve True
        if el == elemento:
            return True
    # Aquí llegamos solo si ya he recorrido toda la lista
    return False
```

```
El tipo del dato que devuelve la función es <class 'tuple'>
El primer y último elemento spn (1, 67)
```

6 Otras características de las funciones

- Descomposición: pienso qué funciones voy a necesitar en mi programa y descompongo el algoritmo en esas funciones. Lo hago aunque mi función solo se vaya a usar una vez
- Ocultación de la implementación: puedo usar una función sin saber cómo está implementada y puedo cambiar su implementación (código) sin cambiar el resto del programa

[20]:
```
# Ejemplo de ocultación de la implementación. Tengo dos formas de
# implementar la función pero las dos se comportan igual externamente

def mi_max(lis : list)-> int:
    """Devuelve el máximo de una lista"""
    # Uso la función max()
    return max(lis)

a = [1, 5, 6, 2, 7, 2, 0, 8]
print("El máximo de la lista", a ,"es", mi_max(a))
```

```
def mi_max(lis : list)-> int:
    """Devuelve el máximo de una lista"""
    # Uso un bucle
    maxim = lis[0]
    for elem in lis:
        if elem > maxim:
            maxim = elem
    return elem

a = [1, 5, 6, 2, 7, 2, 0, 8]
print("El máximo de la lista", a ,"es", mi_max(a))
```

```
El máximo de la lista [1, 5, 6, 2, 7, 2, 0, 8] es 8
El máximo de la lista [1, 5, 6, 2, 7, 2, 0, 8] es 8
```

7 Más sobre parámetros

7.1 Valor por defecto

- Puedo dar un valor por defecto a los parámetros, dándoles un valor en la cabecera de la función
- Los parámetros con valores por defecto también reciben el nombre de parámetros opcionales
- En caso de que tenga parámetros con valor por defecto deben ir siempre detrás de los que no lo tienen

[4]:
```
# Función en la que los dos parámetros tienen valor por defecto
def lanzamiento(inicio = 10, exito = False):
    """Esta función simula el lanzamiento de un cohete"""
    print("Todo listo para el lanzamiento")
    print('Comienza la cuenta atrás')
    for i in range(inicio, -1, -1):
        print(i, end=" ")
    print('¡Despegamos!')
    if exito:
        print("¡Houston, todo bien por aquí!")
        print("Mira la luna, parece un queso")
    else:
        print("¡Houston, Houston, tenemos un problema!")
        print("Abortamos la misión, mañana probaremos otra vez")

# Al tener parámetros con valores por defecto, puedo llamar a la
# función de varias formas
# Sin parámetros: inicio vale 10 y exito vale False
lanzamiento()
# Solamente con un parámetro: inicio vale 10
lanzamiento(exito = False)
# exito vale False
```

```
lanzamiento(inicio = 20)
# sin poner el nombre del parámetro: el valor se asigna al
# parámetro por orden: inicio vale 30 y exito vale False
lanzamiento(30)
```

```
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
Todo listo para el lanzamiento
Comienza la cuenta atrás
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
Todo listo para el lanzamiento
Comienza la cuenta atrás
30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
```

7.2 Empaquetado y desempaquetado de parámetros

- Relación que hay entre tuplas/listas con los parámetros de las funciones
- Operador `*` para parámetros de funciones
- Primer uso: crear funciones con un número variable de parámetros. Sintaxis: `def funcion(*parametro):` Se puede llamar a la función con cualquier número de parámetros. Todos los valores recibidos como parámetro se meten en una tupla. `print()` es un ejemplo de una función de este tipo
- Segundo uso: en una función con varios parámetros, utilizamos una tupla o una lista para dar valor a los parámetros de una sola vez. Coge cada uno de los elementos de la tupla/lista y se los asigna a cada uno de los parámetros en orden. La tupla/lista debe tener el mismo número de elementos que parámetros tiene la función

[15]:
```
# Ejemplo de función con un número variable de parámetros
def suma_cosas(*param: int):
    """ Función que recibe un número indeterminado de parámetros
    y los suma"""
    suma = 0
    # param es una tupla, así que uso un for para recorrerla
    for elem in param:
        suma = suma + elem
```

```
    return suma

# Llamo a la función con un parámetro
a = suma_cosas(3)
print(a)
# La llamo con unos cuantos
a = suma_cosas(5, 3, 5, 8, 8, 0, 12)
print(a)

# Llamando a lanzamiento con dos parámetros
# Este es el modo habitual, doy un valor a cada parámetro
lanzamiento(10, False)
# También puedo usar una tupla
tu = (10, False)
lanzamiento(*tu)
# Lo anterior es equivalente a esto
lanzamiento(tu[0], tu[1])
```

```
3
41
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
```

8 Variables y funciones

- ¿Qué relación hay entre la variable que usamos para dar valor a un parámetro y el parámetro?
- No es necesario que la variable que uso para dar valor a un parámetro se llame como el parámetro

8.1 Paso por valor y paso por referencia

- *Paso por valor*: copio el valor de la variable en el parámetro y trabajo con la copia. Nada de lo que se haga dentro de la función afecta a la variable original. Esta es la forma de pasar variables de Python

- *Paso por referencia*: trabajo directamente con la variable original, cualquier cambio que haga al parámetro dentro de la función se realiza realmente sobre la variable original
- Pasar por referencia o pasar por valor depende del lenguaje. Hay lenguajes que dejan elegir.
- Aunque en Python, como hemos visto, hay paso por valor, cuando el parámetro pertenece a un tipo mutable, se comporta como si fuera paso por referencia: ¡cualquier cambio hecho al parámetro se le hace también a la variable original!
- Receta:
    - Si el parámetro es de un **tipo inmutable**: cualquier cambio hecho dentro de la fucnión no afecta a la variable original (paso por valor)
    - Si el parámetro es de un **tipo mutable** (listas, diccionarios, objetos): los cambios en el parámetro se reflejan en la variable original (equivalente a paso por referencia)
- La razón de este comportamiento es que para los tipos mutables trabajamos con el puntero y al copiar el puntero en el parámetro estamos en realidad trabajando con la variable original (está relacionado con lo que pasaba con la copia superficial y con la profunda)

[17]:
```python
# Uso dos variables para dar valor a los parámetros
# El nombre de la variable y el del parámetro no tiene por qué ser
# el mismo
a = 10
b = False
lanzamiento(a, b)

# Una función que cambia el valor de un parámetro inmutable
def por_dos(parametro: int)-> int:
    """ Multiplica por dos el valor del parámetro recibido y lo
    devuelve """
    parametro = parametro * 2
    return parametro

# Llamo a la función con un literal
a = por_dos(4)
print(a)

# La llamo con una variable
b = 4
a = por_dos(b)
print("El valor del parámetro es", a)
print("El valor de la variable con la que la he llamado es", b)
# Aunque el valor del parámetro ha cambiado, el de la variable
# original no lo ha hecho

# Una función que cambia el valor de un parámetro mutable
def lista_por_dos(param: list) -> list:
    """ Multiplica por dos el valor de cada elemento del parámetro
    recibido y lo devuelve """
    for i in range(len(param)):
```

```
        param[i] = param[i] * 2
    return param

a = [1, 2, 3]
b = lista_por_dos(a)
print("El valor del parámetro es", a)
print("El valor de la variable con la que la he llamado es", b)
```

```
Todo listo para el lanzamiento
Comienza la cuenta atrás
10 9 8 7 6 5 4 3 2 1 0 ¡Despegamos!
¡Houston, Houston, tenemos un problema!
Abortamos la misión, mañana probaremos otra vez
8
El valor del parámetro es 8
El valor de la variable con la que la he llamado es 4
El valor del parámetro es [2, 4, 6]
El valor de la variable con la que la he llamado es [2, 4, 6]
```

8.2 Ámbito de una variable

- Ámbito: la parte de un programa donde puedo usar una variable
- Hasta ahora hemos visto tres tipos de variables: variables normales (las que no están definidas dentro de una función), variables locales (las definidas dentro de una función) y parámetros
- Con variables normales el ámbito va desde que le doy el valor inicial hasta el fin del programa
- El ámbito de variables locales y parámetros es solamente la función. No puedo usar ni un parámetro ni una variable local fuera de la función donde han sido definidos
- Cada función tiene su propio espacio de nombres (*namespace*), puedo reutilizar los nombres de variables locales y parámetros de una función en otra

[19]:
```
# Una función con un parámetro y una variable local
def funcion(para: int)-> int:
    """ Suma 8 al parámetro recibido y lo devuelve"""
    var_local = 8
    para = var_local + para
    return para

# En funcion2 puedo reutilizar los nombres que ya usé en funcion1
# Esto no está mal visto porque pertenecen a dos espacios de nombre
# distintos
def funcion2(para: int)-> int:
    """ Suma 18 al parámetro recibido y lo devuelve"""
    var_local = 18
    para = var_local + para
    return para

a = funcion(18)
```

```
print(a)
b = funcion2(22)
print(b)
# Si intento esto, me da error
#print(var_local)
```

```
26
40
```

8.3 Variables globales

- Variable global: una variable que se puede utilizar en todas las funciones de un programa
- Es muy poco recomendable usarlas porque hacen el código difícil de entender y son propensas a errores
- **Está prohibido usarlas este año**
- En una función puedo (pero no debo) usar una variable definida antes en el programa principal, pero solo en modo lectura. En el momento en que cambie su valor, lo que estoy haciendo es crear una nueva variable local sin relación con la otra
- Si quiero cambiar su valor tengo que declararla como global con la palabra reservada `global`

[20]:
```
# Variables declaradas fuera de las funciones
a = 5
b = 8

# Función que usa las variables anteriores en modo lectura
def func():
    # Puedo usar las variables anteriores aquí, pero solo para
    # leerlas
    print("El valor de a en la 1ª función es", a)
    print("El valor de b en la 1ª función es", b)

# Función que cambia el valor de una variable no local
def func2():
    # En realidad lo que hace es crear una nueva variable llamada
    # también 'a' y asignarle un 6, no trabaja con la externa
    a = 6
    print("El valor de a en la 2ª función es", a)

# Función que define b como global
def func3():
    global b
    # Aquí estamos cambiando el valor de la b externa
    b = 12
    print("El valor de b en la 3ª función es", b)


print("El valor de a es", a)
print("El valor de b es", b)
```

```
print("Ejecutamos la primera función")
func()
print("Tras la 1ª función el valor de a es", a)
print("Tras la 1ª función el valor de b es", b)
print("Ejecutamos la segunda función")
func2()
print("Tras la 2ª función el valor de a es", a)
print("Ejecutamos la tercera función")
func3()
print("Tras la 3ª función el valor de b es", b)
```

```
El valor de a es 5
El valor de b es 8
Ejecutamos la primera función
El valor de a en la 1ª función es 5
El valor de b en la 1ª función es 8
Tras la 1ª función el valor de a es 5
Tras la 1ª función el valor de b es 8
Ejecutamos la segunda función
El valor de a en la 2ª función es 6
Tras la 2ª función el valor de a es 5
Ejecutamos la tercera función
El valor de b en la 3ª función es 12
Tras la 3ª función el valor de b es 12
```

9 Sobrecarga de funciones

- Varias funciones con el mismo nombre pero distintos parámetros
- Python no soporta funciones sobrecargadas
- Si creo varias funciones con el mismo nombre, Python se queda con la última