Funciones

¿Para qué necesito funciones?

Hasta ahora todo lo que hemos hecho han sido breves fragmentos de código Python. Esto puede ser razonable para pequeñas tareas, pero nadie quiere reescribir los fragmentos de código cada vez. Necesitamos una manera de organizar nuestro código en piezas manejables.

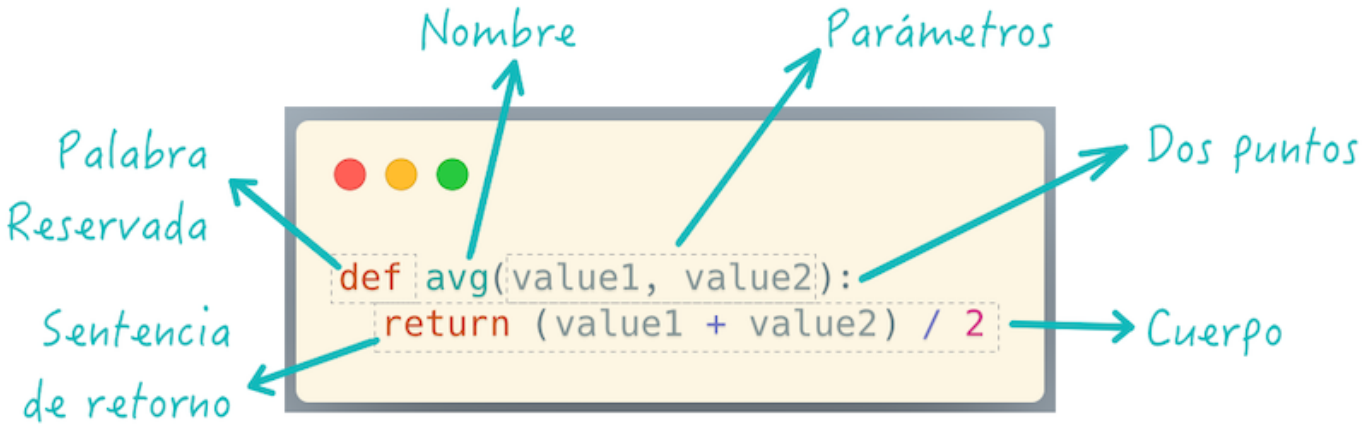
El primer paso para la **reutilización de código** es la **función**. Se trata de un trozo de código con nombre y separado del resto. Puede tomar cualquier número y tipo de *parámetros* y devolver cualquier número y tipo de *resultados*.

Básicamente podemos hacer dos cosas con una función:

- *Definirla* (con cero o más parámetros).
- *Invocarla* (y obtener cero o más resultados).

Definir una función

Para definir una función en Python debemos usar la palabra reservada `def` seguida del nombre de la función, paréntesis rodeando a los parámetros de entrada y finalmente dos puntos `:`



Hagamos una primera función vacía y sin parámetros:

```
def no_hace_nada():
    pass
```

Invocar a una función

Para *invocar* (*llamar*) a una función basta con escribir su nombre y utilizar paréntesis. En el caso de la función sencilla que hemos visto se haría así:

```
1 no_hace_nada()
```

Dado que la función no "hace nada" es razonable que no obtengamos ningún resultado. Vamos a definir otra función que sí tenga algún efecto:

```
1 def haz_un_sonido():
2     print('PI PI')
```

```
1 haz_un_sonido()
```

```
PI PI
```

> Como era de esperar, al invocar a la función obtenemos un mensaje por pantalla, fruto de la ejecución del cuerpo de la función.

Veamos ahora el caso de una función que *retorna* (*devuelve*) algún valor:

```
1 def esVerdadero():
2     return True
```

Podemos hacer uso de esta función, por ejemplo, en sentencias condicionales:

```
1 if esVerdadero():
2     print('🤝')
3 else:
4     print('👎')
```

```
🤝
```

Argumentos y parámetros

Vamos a empezar a crear funciones que reciben parámetros. En este caso escribiremos una función `echo` que recibe el parámetro `anything` y muestra esa variable dos veces separada por un espacio:

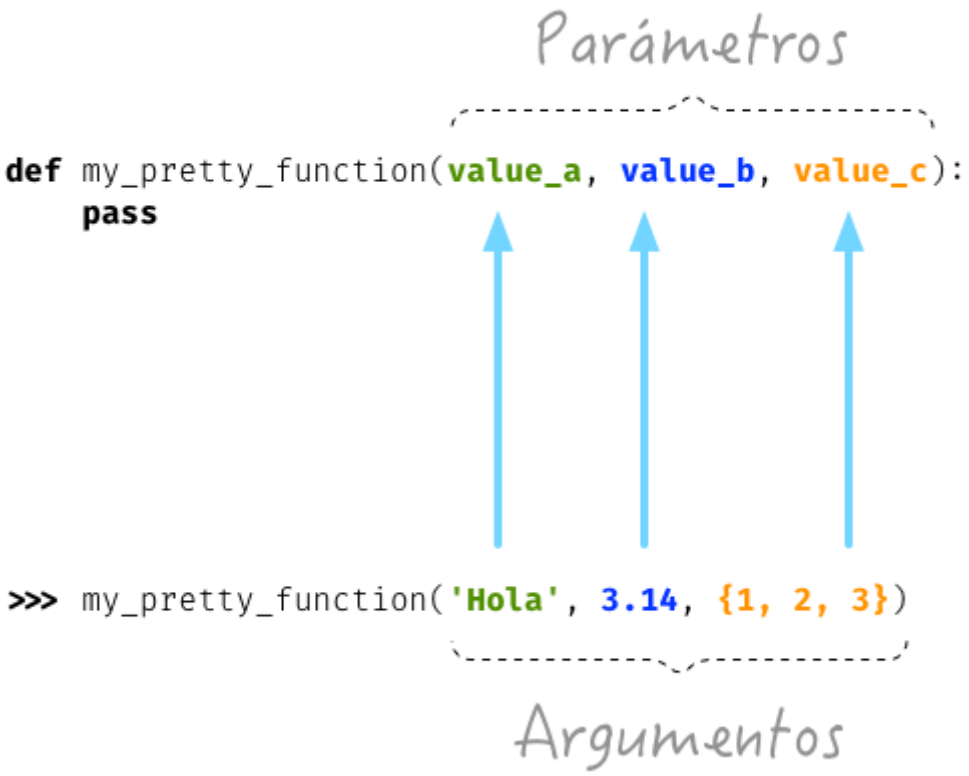
```
1 def echo(anything):
2     return anything + ' ' + anything
```

```
1 echo('Hello world!')
```

```
'Hello world! Hello world!'
```

En este caso, `'Hello world!'` sería un *argumento* de la función.

Cuando llamamos a una función con *argumentos*, los valores de estos argumentos se copian en los correspondientes *parámetros* dentro de la función:



Veamos otra función con algo más de "lógica" en su cuerpo:

```
def fruit_detection(color):
    if color == 'red':
        return "It's an apple"
    elif color == 'yellow':
        return "It's a banana"
    elif color == 'green':
        return "It's a kiwi"
    else:
        return f"I don't know about the color {color}"
```

```
fruit = fruit_detection('green')
fruit
```

```
"It's a kiwi"
```

Aunque una función no tenga un `return` de forma explícita, siempre devolverá `None` de forma implícita:

```
1 print(no_hace_nada())
```

```
None
```

Ejercicio

Escribir una función en Python que reproduzca lo siguiente:

$f(x, y) = x^2 + y^2$

```
1 # Escribe aquí la función:
2 def funcion_de_ejercicio(x, y):
3     return x*x+y*y
4
```

```
1 #Comprueba tu solución:
2 print(funcion_de_ejercicio(3, 4))
```

```
25
```

`None` es útil

`None` es un valor especial de Python que almacena *el valor nulo*. No es lo mismo que `False` aunque lo parezca cuando lo evaluamos como *booleano*:

```
1 thing = None
2 if thing:
3     print("It's some thing")
4 else:
5     print("It's no thing")
```

```
It's no thing
```

Para distinguir `None` del valor booleano `False` se recomienda el uso del operador `is`:

```
1 thing = None
2 if thing is None:
3     print("It's nothing")
4 else:
5     print("It's something")
```

```
It's nothing
```

Vamos a definir una función que imprime si su argumento es `None`, `True` o `False`:

```
def whatis(thing):
    if thing is None:
        print(thing, 'is None')
    elif thing:
        print(thing, 'is True')
    else:
        print(thing, 'is False')
```

Valores que evalúan en booleano como **falsos**:

```
whatis(0)
```

```
0 is False
```

```
whatis(0.0)
```

```
0.0 is False
```

```
whatis('')  # cadena vacía
```

```
 is False
```

```
whatis(())  # tupla vacía
```

```
() is False
```

```
whatis([])  # lista vacía
```

```
[] is False
```

```
whatis({})  # diccionario vacío
```

```
{} is False
```

Valores que evalúan en booleano como **verdaderos**:

```
whatis(0.00001)
```

```
1e-05 is True
```

```
whatis([0])
```

```
[0] is True
```

```
1 whatis([''])
```

```
    [''] is True
```

```
1 whatis(' ')
```

```
      is True
```

Especificar parámetros con valores por defecto

Es posible especificar valores por defecto en los parámetros de una función. El valor por defecto se usará cuando en la llamada a la función no se haya proporcionado el correspondiente argumento.

```
1 def menu(wine, entree, dessert='Tiramisú'):
2     return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

```
1 # Hacemos uso del valor por defecto del parámetro "dessert"
2
3 menu('Ignios', 'Ensalada')
```

```
    {'wine': 'Ignios', 'entree': 'Ensalada', 'dessert': 'Tiramisú'}
```

```
1 # "Sobreescribimos" el valor de "dessert" especificando uno concreto
2
3 menu('Tajinaste', 'Revuelto de setas', 'Helado')
```

```
    {'wine': 'Tajinaste', 'entree': 'Revuelto de setas', 'dessert': 'Helado'}
```

> Los valores por defecto en los parámetros se calculan cuando se **define** la función, no cuando se **ejecuta**.

En la siguiente función, uno esperaría que `result` tuviera una lista vacía en cada ejecución, pero como estamos modificando ese parámetro dentro de la función, este cambio perdura en el tiempo:

```
1 def buggy(arg, result=[]):
2     result.append(arg)
3     print(result)
```

```
1 buggy('a')
```

```
    ['a']
```

```
1 buggy('b')  # se esperaría ['b']
```

```
['a', 'b']
```

Habría funcionado si hubiéramos escrito algo así:

```
1 def works(arg):
2     result = []
3     result.append(arg)
4     return result
```

```
1 works('a')
```

```
['a']
```

```
1 works('b')
```

```
['b']
```

La forma de arreglar el código anterior utilizando un parámetro con valor por defecto sería indicar cuál es la primera llamada:

```
1 def nonbuggy(arg, result=None):
2     if result is None:
3         result = []
4     result.append(arg)
5     print(result)
```

```
1 nonbuggy('a')
```

```
['a']
```

```
1 nonbuggy('b')
```

```
['b']
```

> Esto suele ser pregunta para entrevistas de trabajo en Python!

Reunir/Desplegar argumentos posicionales

Python ofrece la posibilidad de utilizar un asterisco `*` en los parámetros de las funciones. Sirve para **reunir** múltiples argumentos posicionales en una única tupla como valor del parámetro.

```
def print_args(*args):
    print('Positional tuple:', args)
```

Si llamamos a la función sin argumentos no obtendremos nada en `*args`:

```
print_args()
```

```
Positional tuple: ()
```

Pero la parte interesante es que podemos pasar cualquier número de argumentos:

```
print_args(1, 2, 3, 'pescado', 'salado', 'es')
```

```
Positional tuple: (1, 2, 3, 'pescado', 'salado', 'es')
```

También podemos utilizar esta estrategia para establecer en una función una serie de parámetros como *requeridos* y recibir el *resto de argumentos* como opcionales y empaquetados:

```
def sum_all(v1, v2, *args):
    total = 0
    for value in (v1, v2) + args:  # args es una tupla
        total += value
    return total
```

```
sum_all()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-6e0e70053a1a> in <module>
----> 1 sum_all()

TypeError: sum_all() missing 2 required positional arguments: 'v1' and 'v2'
```

SEARCH STACK OVERFLOW

```
sum_all(1, 2)
```

```
3
```

```
sum_all(5, 9, 3, 8, 11, 21)
```

```
57
```

Existe la posibilidad de usar el asterisco `*` en la llamada a la función para **desplegar** los argumentos posicionales:

```
def print_args(*args):
    print('Positional tuple:', args)
```

```
print_args(2, 5, 7, 'x')
```

```
Positional tuple: (2, 5, 7, 'x')
```

```
args = (2, 5, 7, 'x')
```

```
print_args(args)
```

```
Positional tuple: ((2, 5, 7, 'x'),)
```

```
print_args(*args)  # despliegue de argumentos
```

```
Positional tuple: (2, 5, 7, 'x')
```

Documentación

Podemos (y en muchos casos *debemos*) adjuntar **documentación** a la definición de una función incluyendo una cadena de texto (**`docstring`**) en el comienzo de su cuerpo:

```
def echo(anything):
    'echo returns its input argument'
    return anything
```

Podemos escribir un `docstring` con más información utilizando triples comillas `'''` :

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    '''
    if check:
        print(thing)
```

Para imprimir el `docstring` de una función, basta con hacer uso de la función `help` :

```
help(echo)
```

```
Help on function echo in module __main__:

echo(anything)
    echo returns its input argument
```

Si queremos ver el `docstring` en *crudo* sin ningún formato, haríamos lo siguiente:

```
1 print(print_if_true.__doc__)
```

```
        Prints the first argument if a second argument is true.
        The operation is:
            1. Check whether the *second* argument is true.
            2. If it is, print the *first* argument.
```

Las funciones también son objetos

Como ya se ha comentado, en Python "*todo es un objeto*", y también ocurre con las *funciones*. Podemos asignar una función a una variable, podemos usarlas como argumentos de otras funciones y como valor de retorno. Esto permite una gran flexibilidad y aporta nuevas posibilidades al lenguaje.

```
1 def respuesta():
2     print(42)
3
4 respuesta()
```

```
    42
```

Ahora vamos a definir una función que recibe otra función como parámetro y se encarga de invocarla:

```
1 def run_something(func):
2     func()
3
4 run_something(respuesta)  # función "respuesta" como parámetro
```

```
    42
```

Veamos ahora otro ejemplo definiendo una función con argumentos:

```
1 def suma_args(arg1, arg2):
2     print(arg1 + arg2)
```

```
1 def funcion_auxiliar(func, arg1, arg2):
2     func(arg1, arg2)
```

Ahora podemos invocar a la función pasando como parámetros la otra función y dos valores que sumar:

```
1 type(suma_args)
```

```
function
```

```
1 funcion_auxiliar(suma_args, 5, 9)
```

```
14
```

Variables globales

Hasta ahora, hemos estado creando variables dentro de funciones, pero no hemos creado variables fuera de la función. Estas se llaman variables globales.
Intentemos ver qué devuelve `printer1`:

```
1 # Ejemplo de variable global
2
3 artist = "Michael Jackson"
4 def printer1(artist):
5     internal_var1 = artist
6     print(artist, "es un artista")
7
8 printer1(artist)
9 # Mira que pasa si corres la siguiente línea
10 # printer1(internal_var1)
```

Ejercicios de Funciones

Crea una función que divida la primera entrada por la segunda entrada:

```
1 # Crea una función que divida la primera entrada por la segunda entrada:
2
```

Utiliza la función `con` para la siguiente pregunta.

```
1 def con(a, b):
2     return(a + b)
```

¿Se puede usar la función `con` que definimos antes para sumar tanto dos enteros como dos strings?

```
1 # Pruebalo
2
```

¿Se puede usar la función `con` que definimos antes para concatenar listas?

```
1 # Pruebalo
2
```