

# Función

## 1. Declaración

La declaración de una función es muy sencilla, tal y como hemos visto en el capítulo anterior. Basta con utilizar la palabra clave `def`, seguida del nombre que se quiere dar a la función y paréntesis de apertura y cierre que pueden, si es preciso, contener una lista de argumentos, y los dos puntos.

Se abre, de este modo, un bloque que posee su propio espacio de nombres local y que contiene las instrucciones de la función. Termina devolviendo una variable y, si no se indica explícitamente mediante alguna instrucción, la función devuelve `None`.

El nombre de la función debe ser, preferentemente, un nombre representativo de esta. Este nombre es también el nombre de la variable (continente) cuyo valor es la función, que es un objeto (contenido).

Si ya existe una variable con el nombre de la función, se reemplaza por la función, exactamente de la misma manera que cuando se realiza una operación de asignación.

He aquí una función vacía:

```
>>> def f():
...     pass
...
```

He aquí los atributos o métodos del objeto función:

```
>>> list(set(dir(f))-set(dir(object)))
['__module__', '__defaults__', '__annotations__',
 '__kwdefaults__', '__globals__', '__call__', '__closure__',
 '__dict__', '__name__', '__code__', '__get__']
```

Una función está vinculada con el nombre del módulo que contiene su definición:

```
>>> f.__module__
'__main__'
```

Vemos que lleva su mismo nombre:

```
>>> f.__name__
'f'
```

Esta característica es propia de la función y no del nombre de la variable:

```
>>> g = f
>>> g.__name__
'f'
```

He aquí la misma función definida con un docstring:

```
>>> def f():
...     """Docstring útil"""
...
```

La palabra clave `pass` ya no es necesaria, pues la función contiene una instrucción que es este docstring. Forma parte de la documentación del código, y resulta útil para aquellos que deban utilizarla, permitiendo realizar pruebas unitarias.

## 2. Parámetros

### a. Firma de una función

Los dos elementos que constituyen una función son el bloque que contiene su código y su firma, es decir, su nombre seguido de sus parámetros y sus características. Esta firma determina la visibilidad que tienen los elementos exteriores cuando se invoca a la función.

Esta firma encuentra una traducción visible si se analiza el objeto función. De este modo, una función sin parámetros, como la definida anteriormente, dispone de los siguientes atributos:

```
>>> f.__defaults__
>>> f.__kwdefaults__
>>> f.__annotations__
{}
```

Su firma es: `f()`

He aquí una función que recibe tres parámetros:

```
>>> def f(a, b, c):
...     return a + b + c
...
```

Su firma es `f(a, b, c)`

### b. Noción de argumento o de parámetro

Cuando un argumento o parámetro (se aceptan ambas terminologías) está presente en la firma de una función debe, obligatoriamente, recibir un valor. La función que acabamos de escribir debería invocarse de la siguiente manera:

```
>>> f(1, 2, 3)
6
```

Si falta algún argumento o si recibe más de lo esperado, se genera una excepción:

```
>>> f(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 3 arguments (2 given)
```

Estas verificaciones las realiza Python de forma automática y se encarga de gestionar correctamente los espacios de nombres para integrar los valores transmitidos durante la llamada a los nombres de las variables definidas durante la definición de la función:

```
>>> def f(a, b, c):
...     print(locals())
...     return a + b + c
...
>>> f(1, 2, 3)
{'a': 1, 'c': 3, 'b': 2}
6
```

A este respecto, Python permite una transparencia apreciable.

#### c. Valor por defecto

Una firma como la que hemos visto antes implica precisar, para cada llamada de la función, un conjunto de parámetros. O bien, si se quiere simplificar una llamada a la función dejando que no sea obligatorio informar ciertos parámetros, es posible darles un valor por defecto.

Se dice que estos parámetros son opcionales y su declaración es muy parecida a la de los parámetros obligatorios, indicando simplemente su valor por defecto en la firma de la función:

```
>>> def f(a=0, b=0, c=0):
...     print(locals())
...
```

Con cada llamada, los parámetros que se pasan a la función se asignan a las variables correspondientes, mientras que otros reciben su valor por defecto:

```
>>> f()
{'a': 0, 'b': 0, 'c': 0}
>>> f(1)
{'a': 1, 'b': 0, 'c': 0}
>>> f(1, 2)
{'a': 1, 'b': 2, 'c': 0}
>>> f(1, 2, 3)
{'a': 1, 'b': 2, 'c': 3}
```

No obstante, si se pasan demasiados parámetros, se produce un error:

```
>>> f(1, 2, 3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes at most 3 positional arguments (4 given)
```

La manifestación evidente de este funcionamiento se encuentra en un atributo del objeto función que contiene la lista de valores por defecto:

```
>>> f.__defaults__
(0, 0, 0)
```

El elemento esencial en la firma de una función es el orden en que se declaran los parámetros. Este orden determina el valor de cada variable. De este modo, pueden convivir los parámetros obligatorios y opcionales:

```
>>> def f(a, b, c=0):
...     print(locals())
...
>>> f(1, 2)
{'a': 1, 'b': 2, 'c': 0}
>>> f(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes at least 2 arguments (1 given)
```

Aun así, es primordial respetar el orden, y no hay más que un valor por defecto. De este modo:

```
>>> f.__defaults__
(0,)
```

Una firma como `f(a, b=42, c)` no es correcta, pues si se invocara a la función con dos parámetros, el último parámetro no estaría informado, provocando un error incluso aunque el segundo parámetro ve cómo su valor se sustituye por el de la llamada.

La lógica implícita a la firma de las funciones obliga a situar los parámetros opcionales después de los parámetros obligatorios.

#### d. Parámetros nombrados

Cuando se tiene varios parámetrosopcionales, es posible modificar el valor por defecto de uno de ellos sin estar obligado a tener que pasar los valores por defecto de los parámetros anteriores. Por ejemplo:

```
>>> def f(a=0, b=0, c=0):
...     print(locals())
...
```

Para modificar el valor de `b` sin afectar a `a`, es posible informar el valor por defecto de `a` en la llamada:

```
>>> f(0, 4)
{'a': 0, 'c': 0, 'b': 4}
```

Todos los lenguajes lo permiten, pero algunos como Python permiten, también, pasar únicamente el valor que hay que modificar, dándole nombre en la llamada:

```
>>> f(b=4)
{'a': 0, 'c': 0, 'b': 4}
```

Es posible nombrar las variables durante la llamada, tanto si el parámetro es obligatorio como si es opcional, y es posible utilizar de manera conjunta parámetros nombrados y parámetros no nombrados.

Las siguientes instrucciones son equivalentes:

```
>>> f(1, 2, 3)
6
>>> f(a=1, b=2, c=3)
6
>>> f(b=2, a=1, c=3)
6
>>> f(1, 2, c=3)
6
```

Preste atención, no obstante, a los parámetros no nombrados que deben pasarse en primer lugar:

```
>>> f(a=1, 2, 3)
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Preste atención también a no declarar varias veces la misma variable:

```
>>> f(1, a=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
```

#### e. Declaración de parámetros extensibles

Una de las características esenciales de Python es que tiene en cuenta, de forma sencilla y limpia, un número variable de argumentos en la firma de una función.

De este modo es posible agrupar los argumentos no nombrados en una n-tupla y los argumentos nombrados en un diccionario.

He aquí la forma de recuperar los argumentos no nombrados:

```
>>> def f(*args):
...     return locals()
...
>>> f(1, 2, 3, 4, 5, 6)
{'args': (1, 2, 3, 4, 5, 6)}
```

He aquí la forma de recuperar los argumentos nombrados:

```
>>> def f(**kwargs):
...     return locals()
...
>>> f(a=1, b=2, c=3)
{'kwargs': {'a': 1, 'c': 3, 'b': 2}}
```

En ambos casos, `args` y `kwargs` son variables locales de la función, que se utilizan respectivamente como tupla o como diccionario. En este último caso las variables nombradas pueden agregarse, de manera unitaria, al espacio de nombres local de la función:

```
>>> def f(**kwargs):
...     locals().update(**kwargs)
...     del kwargs
...     return locals()
...
>>> f(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
```

Para los argumentos no nombrados no es posible hacer esto, a menos que se fijen aleatoriamente los nombres de las variables, lo cual resulta poco interesante.

He aquí un ejemplo que utiliza todos los tipos de parámetros en su firma:

```
>>> def f(a, b=0, *args, **kwargs):
...     return a + b + sum(args) + sum(kwargs.values())
...
>>> f(1, 2, 3, 4, y=5, z=6)
21
```

Aun así, el orden de los argumentos entre sí es extremadamente importante. Los argumentos obligatorios se informan en primer lugar, a continuación vienen los argumentosopcionales y en último lugar los parámetros extensibles. Entre ellos, los argumentos no nombrados se informan en primer lugar y los argumentos nombrados (`kwargs`) se sitúan, obligatoriamente, en último lugar.

En este caso concreto, `a` vale 1, `b` vale 2, mientras que 3 y 4 son argumentos no nombrados presentes en la lista `args` e `y` y `z` son argumentos nombrados almacenados en el diccionario `kwargs`. El único parámetro obligatorio es `a`.

He aquí una función que permite ver los detalles:

```
>>> def f(a, b=0, *args, **kwargs):
...     print('a=%s' % a)
...     print('b=%s' % b)
...     print('args=%s' % str(args))
...     print('kwargs=%s' % str(kwargs))
...
>>> f(1, 2, 3, 4, y=5, z=6)
a=1
b=2
args=(3, 4)
kwargs={'y': 5, 'z': 6}
```

Dicha función recibe un parámetro obligatorio, y todos los demás son opcionales (**b** tiene un valor por defecto y los atributos con asterisco o doble asterisco son opcionales por definición).

#### f. Paso de parámetros con asterisco

Durante la llamada a la función, es posible pasar los argumentos no nombrados mediante una secuencia prefijada por un asterisco y los argumentos nombrados mediante un diccionario prefijado por dos asteriscos:

```
>>> f(*[1, 2, 3, 4], **{'y': 5, 'z': 6})
a=1
b=2
args=(3, 4)
kwargs={'y': 5, 'z': 6}
```

Esta notación prefijada por uno o dos asteriscos se utiliza en otros contextos (no necesariamente en la firma de una función) y permiten pasar, de manera muy sencilla, una lista a una enumeración de valores no nombrados y un diccionario a una enumeración de valores nombrados.

Esta flexibilidad es una de las principales armas de Python y resulta una herramienta esencial para producir un código genérico y extensible de manera sencilla.

#### g. Firma universal

Es fácil producir una firma que acepte todo tipo de parámetros, pasados de cualquier manera:

```
>>> def f(*args, **kwargs):
...     return sum(args) + sum(kwargs.values())
...
>>> f(1, 2, 3, 4, y=5, z=6)
21
```

Este tipo de función resulta ultraflexible, aunque requiere procesar los datos recibidos a continuación. Si algunos parámetros son obligatorios, es preciso declararlos como parámetros obligatorios.

La mejor forma de diseñar la firma de una función es pensar en las formas en las que se la querrá invocar. Hay

que pensar, también, que esta firma puede evolucionar y que su evolución debería realizarse de manera que las antiguas llamadas a la función sigan siendo válidas. De este modo, una evolución de la función debería mantener una firma compatible con la antigua.

Esta técnica se utiliza también para permitir, durante la llamada a la función, pasar sin hacer distinción toda una serie de datos. Es la firma la que permite ordenar los datos y vincular aquellos que necesita, dejando los demás en parámetros extendidos que sirvan como «papelera».

Una llamada a la función autoriza a pasar más parámetros que los realmente necesarios sin producir, por ello, un error. He aquí un ejemplo:

```
>>> def f1(a, b, *args, **kwargs):
...     return locals()
...
>>> def f2(b, c, *args, **kwargs):
...     return locals()
...
>>> f1(**datas)
{'a': 1, 'args': (), 'b': 2, 'kwargs': {'c': 2}}
>>> f2(**datas)
{'c': 2, 'args': (), 'b': 2, 'kwargs': {'a': 1}}
```

Este procedimiento se utiliza a menudo y resulta práctico en ocasiones, aunque una vez más diremos que definir una firma restrictiva es un medio de automatizar todo un trabajo de verificación del correcto paso de argumentos y prevenir potenciales problemas de desarrollo. Los parámetros con asterisco no deberían utilizarse sistemáticamente sustituyendo a los parámetros clásicos.

#### **h. Obligar a un parámetro a ser nombrado (keyword-only)**

Por ciertos motivos, vinculados generalmente a razones de legibilidad en las llamadas a las funciones, puede迫使 un parámetro a ser nombrado. He aquí un ejemplo de función:

```
>>> def f(a, b, operador):
...     pass
...
```

La llamada a la función no es muy explícita, y para comprender la firma es necesario comprender el código:

```
>>> f(1, 2, '+')
```

Para corregirlo, en la firma de la función, basta con ubicar el parámetro detrás de `*args`:

```
>>> def f(a, b, *args, operador):
...     pass
...
```

Si se invoca a la función como se ha hecho antes, se obtendrá un error:

```
>>> f(1, 2, '+')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() needs keyword-only argument operador
```

La siguiente llamada resulta más clara. Ciertos números de funciones y métodos de la librería Python 3 ya utilizan este procedimiento.

```
>>> f(1, 2, operador='+')
```

Esta forma de proceder permite, a su vez, facilitar un posible cambio de firma en las versiones superiores de la función manteniendo la compatibilidad con la versión anterior.

Por otro lado, es posible asignar un valor por defecto a un parámetro y los parámetros obligatorios deben declararse antes de los parámetros nombrados opcionales, siendo coherente con los parámetros clásicos:

```
>>> def f(a, b, *args, operador='+'):
...     pass
...
```

Los parámetros nombrados no tienen relación de orden entre ellos, como hemos visto anteriormente. En consecuencia, lo siguiente es posible y tiene sentido:

```
>>> def f(a, b='', *args, x=0, y):
...     pass
...
```

He aquí una posible llamada a la función anterior:

```
>>> f(1, 2, 3, y='')
```

En este caso, **a** vale 1, **b** vale 2, **args**, los parámetros nombrados **x** e **y** valen, respectivamente, 0 y una cadena vacía.

La mínima llamada es una de las siguientes:

```
>>> f('valor de a', y='valor de y')
>>> f(a='valor de a', y='valor de y')
```

O incluso utilizando parámetros con doble asterisco:

```
>>> f(**{'a': 'a', 'y': 'y'})
```

El hecho de que dicho parámetro tenga un valor por defecto se ve en un atributo del objeto función particular:

```
>>> f.__kwdefaults__  
{'x': 0}
```

Mientras que los demás parámetros clásicos ven sus valores por defecto almacenados en otro atributo ya presente:

```
>>> f.__defaults__  
('',)
```

Observe las diferencias en la representación, pues en el primer caso es el orden de los argumentos lo que importa, y se utiliza una tupla, mientras que en el segundo caso es el nombre del parámetro lo que cuenta y se utiliza un diccionario, donde las claves representan los nombres de los parámetros

Una vez más, no hay magia y estos atributos del objeto son modificables:

```
>>> f.__defaults__ = (1, 2)  
>>> f.__kwdefaults__ = {'x': 'x', 'y': 'y'}
```

Con estas modificaciones es posible invocar a la función sin parámetros, pues acabamos de darles, a todos, un valor por defecto:

```
>>> f()
```

## i. Anotaciones

El tipado estático hace que la firma de una función incluya el tipo de las variables. En Python, no es el caso, pues el tipado es dinámico.

Para invocar a una función, no existe ninguna manera de verificar, antes de la ejecución del código, que los tipos esperados están bien pasados. Esto aporta cierta flexibilidad, pues es posible utilizar la misma firma para gestionar varios casos.

Por ejemplo, no es extraño encontrar funciones que esperan recibir como parámetro un flujo de datos y que son capaces de trabajar indistintamente con un buffer, el descriptor de un archivo o incluso una simple cadena de caracteres que contiene la ruta hacia un archivo.

Este tipo de necesidades puede aparecer en varias funciones, incluso es posible crear un decorador que se encargue de tener en cuenta los distintos casos posibles para devolver un tipo único a la función a la que se aplique. Procediendo así, la funcionalidad del decorador se capitaliza y reutiliza en otras funciones.

Por el contrario, puede resultar útil verificar el tipo de los parámetros, bien el tipo devuelto o incluso su propio tipo. Esto puede realizarse fácilmente mediante decoradores (consulte la sección Decorador del capítulo Patrones de diseño). No obstante, este mecanismo requiere escribir estos decoradores y no es fácil, ni mucho menos rápido, construirlos de forma genérica y reutilizable.

Aun así, Python proporciona un nuevo mecanismo que complementa esto: las anotaciones

(<http://www.python.org/dev/peps/pep-3107/>). Ahora es posible precisar el tipo de los datos esperados, así como el tipo del resultado, utilizando anotaciones directamente en la firma de la función.

Preste atención, por un lado, a que Python no vuelve al principio de duck typing, sino que permite a sus desarrolladores implementar una verificación de los tipos de datos que se pasan como parámetro. Por otro lado, este mecanismo no tiene nada que ver con lo que hacen otros lenguajes con tipado estático. Es importante no confundir ambas nociones.

He aquí dicha declaración:

```
>>> def f(a:str, b:int)->int:  
...     print(locals())  
...     return 1  
...
```

Las anotaciones, por sí mismas, no garantizan que los tipos se respeten:

```
>>> f(1, 2)  
{'a': 1, 'b': 2}  
1  
>>> f('', 2)  
{'a': '', 'b': 2}  
1
```

Lo importante para las anotaciones es que el usuario de una función pueda saber lo que se espera como parámetro. Esto se realiza fácilmente:

```
>>> f.__annotations__  
{'a': <class 'str'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

A continuación, es fácil realizar un decorador en dos niveles adaptados a una firma de función específica, pero para ello es necesario que la firma del decorador del segundo nivel sea idéntica a la del decorador del primer nivel.

He aquí un ejemplo donde se destacan en negrita las firmas del decorador de segundo nivel y de la función decorada:

```
>>> def wrapper(f):  
...     def wrapped(a, b=42):  
...         if type(a) != str:  
...             raise TypeError("El argumento a debería ser  
de tipo <class 'str'> y es de tipo %s" % type(a))  
...         if 'b' not in locals():  
...             b = f.__defaults__.get(b)  
...         if type(b) != int:  
...             raise TypeError("El argumento b debería ser  
de tipo <class 'int'> y es de tipo %s" % type(b))  
...         r = f(a, b)  
...         if type(r) != int:  
...             raise TypeError("El tipo del resultado
```

```
debería ser <class 'int'> y es de tipo %s" % type(r))
...
    return r
...
    return wrapped
...
>>> @wrapper
... def f(a:str, b:int=42)->int:
...     print('f', locals())
...     return 1
...

```

El decorador es específico de la función, pues recupera la firma de esta última. No es genérico, aunque con el decorador por un lado y las anotaciones por otro podemos llegar a disponer de las herramientas necesarias para realizar una verificación de tipos sobre los argumentos y el resultado de la función.

Tenemos lo siguiente:

```
>>> f('', 1)
f {'a': '', 'b': 1}
1
>>> f('')
f {'a': '', 'b': 42}
1
>>> f(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in wrapped
TypeError: El argumento a debería ser de tipo <class 'str'> y es
de tipo <class 'int'>
```

Una llamada con los tipos incorrectos provoca una excepción que indica que, o bien alguno de los parámetros, o bien el resultado, no están conformes.

Se trata de una forma de imponer precondiciones y poscondiciones.

No obstante, el hecho de que este mecanismo no sea genérico y que no exista la posibilidad de interconectar el decorador con las anotaciones limita el principio y la genericidad.

Crear un decorador genérico implica tener una firma de función genérica para el decorador de segundo nivel, es decir, una firma `wrapped(*args, **kwargs)`. O bien, el uso de `*args` nos priva del nombre de la variable a la que se asocia cada valor contenido. Esto nos priva, por tanto, de toda información explotable para poder establecer el vínculo con anotaciones. La llamada a la función debe nombrar todos los parámetros.

Una vez constatado esto, resulta fácil construir un decorador genérico aplicable a cualquier función cuya firma sea `f(*args, ...)`, respetando las reglas que hemos visto antes. Por el contrario, nada impide anotar únicamente parte de los parámetros.

El decorador enumerará el conjunto de anotaciones, con la excepción de `result`, reservada al resultado de la función, y buscará entre los parámetros que se pasan durante la llamada de la función o, en su defecto, en los parámetros por defecto si el tipo es correcto.

A continuación, realizará la misma operación sobre el resultado si existe la anotación correspondiente en la firma de la función decorada.

```
>>> def wrapper(f):
...     def wrapped(*args, **kw):
...         for n, t in f.__annotations__.items():
...             if n == 'return':
...                 continue
...             a = type(kw.get(n, f.__kwdefaults__ != None and f.__kwdefaults__.get(n) or None))
...             if a != t:
...                 raise TypeError("El argumento %s
debería ser de tipo %s y es de tipo %s" % (n, t, type(a)))
...             r = f(**kw)
...             if 'result' in f.__annotations__:
...                 if type(r) != f.__annotation__['result']:
...                     raise TypeError("El tipo del
resultado debería ser %s y es de tipo %s" %
(f.__annotations__['result'], type(r)))
...             return r
...     return wrapped
...
>>> @wrapper
... def f(*args, a:str, b:int=42)->int:
...     print('f', locals())
...     return 1
...
>>> f(a='', b=1)
f {'a': '', 'args': (), 'b': 1}
1
>>> f(a='')
f {'a': '', 'args': (), 'b': 42}
1
>>> f(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in wrapped
TypeError: El argumento a debería ser de tipo <class 'str'> y es
de tipo <class 'type'>
```

