Python

Guía para ser un Pythonista

Juan José Lozano Gómez – j2logo.com

13 Funciones

Índice de contenidos

Introducción

Las funciones en Python, y en cualquier lenguaje de programación, son estructuras esenciales de código. Una función es un grupo de instrucciones que constituyen una unidad lógica del programa y resuelven un problema muy concreto.

Este tema te explica en detalle cómo definir una función en Python, qué particularidades tienen en el lenguaje y cómo usarlas en tus aplicaciones.

Qué es una función

Como te decía en la introducción, una función es un conjunto de instrucciones que constituyen una unidad lógica de un programa y tienen un doble objetivo:

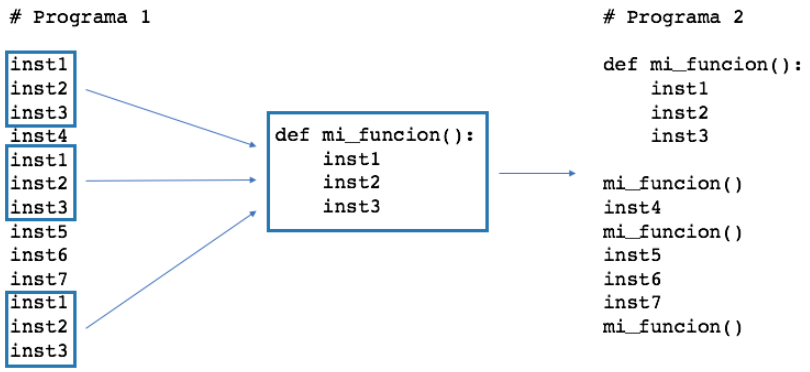
- Dividir y organizar el código en partes más sencillas.
- Encapsular el código que se repite a lo largo de un programa para ser reutilizado.

Python define de serie un conjunto de funciones que podemos utilizar directamente en nuestras aplicaciones. Algunas de ellas las has visto en temas anteriores. Por ejemplo, la función `len()`, que obtiene el número de elementos de un objeto contenedor como una *lista*, una *tupla*, un *diccionario* o un *conjunto*. También hemos visto la función `print()`, que muestra por consola un texto.

¿Dónde están definidas las funciones como `len()` o `print()`? ¿Por qué tenemos acceso a ellas directamente? La respuesta a estas preguntas te la daré en el tema siguiente.

Sin embargo, tú puedes definir tus propias funciones para estructurar el código de manera que sea más legible y/o puedas reutilizar aquellas partes que se repiten a lo largo de una aplicación. Esto es una tarea fundamental a medida que va creciendo el número de líneas de un programa.

La idea la puedes observar en la siguiente imagen:



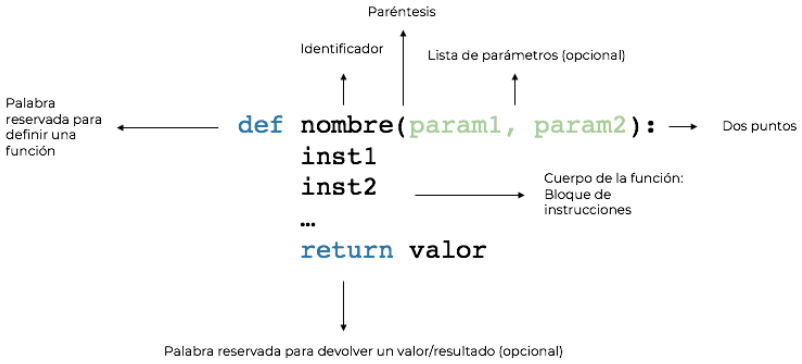
En principio, un programa (*Programa 1*) es una secuencia ordenada de instrucciones que se ejecutan una a continuación de la otra. Sin embargo, cuando se utilizan funciones, puedes agrupar parte de esas instrucciones como una unidad lógica independiente que puedes llamar en cualquier momento (*Programa 2*).

Además, como resultado de ejecutar su bloque de instrucciones, **es muy común que una función devuelva un valor.**

Puedes ver una función como una nueva instrucción o sentencia que añades al lenguaje, pero que solo está disponible para el programa en que se define.

Cómo definir una función en Python

La siguiente imagen muestra el esquema de una función en Python:



Como puedes observar, para definir una función en Python se utiliza la palabra reservada `def`. A continuación, viene el nombre o identificador de la función que es el que se utiliza para invocarla. Después del nombre hay que incluir los paréntesis y una lista opcional de parámetros. Por último, la cabecera o definición de la función termina con dos puntos.

Tras los dos puntos se incluye el cuerpo de la función (con un sangrado mayor) que no es más que el conjunto de instrucciones que se encapsulan en dicha función y que le dan significado.

En último lugar y de manera opcional, se añade la sentencia `return` para devolver un resultado.

IMPORTANTE: Cuando la primera instrucción de una función es un string encerrado entre tres comillas simples ''' o dobles """, a dicha instrucción se le conoce como docstring. El docstring, como ya hemos mencionado anteriormente, es una cadena que se utiliza para documentar la función, es decir, indicar qué hace dicha función.

Cómo usar o llamar a una función

Para usar o invocar a una función, simplemente hay que escribir su nombre en el programa como si de una instrucción más se tratara. Eso sí, pasando los argumentos necesarios según los parámetros que defina la función.

Veámoslo con un ejemplo. El siguiente programa define una función que muestra por pantalla el resultado de multiplicar un número por cinco:

```
def multiplica_por_5(n):
    print(f'{n} * 5 = {n * 5}')

print('Comienzo del programa')
multiplica_por_5(7)
print('Siguiente')
multiplica_por_5(113)
print('Fin')
```

La función `multiplica_por_5(n)` define un parámetro llamado `n` que es el que se utiliza para multiplicar por 5. El resultado del programa anterior sería el siguiente:

```
Comienzo del programa
7 * 5 = 35
Siguiente
113 * 5 = 565
Fin
```

Como puedes observar, el programa comienza su ejecución en la *línea 4* (`print('Comienzo...')`) y va ejecutando las instrucciones una a una de manera ordenada. Cuando se encuentra el nombre de la función `multiplica_por_5()`, el flujo de ejecución pasa a la primera instrucción de la función. Cuando se llega a la última instrucción de la función, el flujo del programa sigue por la instrucción que hay a continuación de la llamada de la función.

IMPORTANTE: Diferencia entre parámetro y argumento. La función `multiplica_por_5(n)` define un parámetro llamado `n`. Sin embargo, cuando desde el código se invoca a la función, por ejemplo, `multiplica_por_5(7)`, se dice que se llama a la función con el argumento `7`.

Sentencia return

En la sección anterior te indicaba que cuando acaba la última instrucción de una función, el flujo del programa continúa por la instrucción que sigue a la llamada a dicha función. Hay una excepción: usar la sentencia `return`. `return` hace que termine la ejecución de la función cuando aparece y el programa continúa por su flujo normal.

Además, `return` se suele utilizar para devolver un valor.

La sentencia `return` es opcional, puede devolver, o no, un valor y es posible que aparezca más de una vez dentro de una misma función.

A continuación, te muestro varios ejemplos de uso de `return`:

return que no devuelve ningún valor

La siguiente función muestra por pantalla el cuadrado de un número solo si este es par:

```
>>> def cuadrado_de_par(numero):
...     if not numero % 2 == 0:
...         return
...     else:
...         print(numero ** 2)
...
>>> cuadrado_de_par(8)
64
>>> cuadrado_de_par(3)
```

Varios return en una misma función

La función `es_par()` devuelve `True` si un número es par y `False` en caso contrario:

```
>>> def es_par(numero):
...     if numero % 2 == 0:
...         return True
...     else:
...         return False
...
>>> es_par(2)
True
>>> es_par(5)
False
```

Devolver más de un valor con return en Python

En Python, es posible devolver más de un valor con una sola sentencia `return`. Por defecto, con `return` se puede devolver una tupla de valores. Un ejemplo sería la siguiente función `cuadrado_y_cubo()` que devuelve el cuadrado y el cubo de un número:

```
>>> def cuadrado_y_cubo(n):
...     return n ** 2, numenro ** 3
...
>>> cuad, cubo = cuadrado_y_cubo(4)
>>> cuad
16
>>> cubo
64
```

Sin embargo, se puede usar otra técnica devolviendo los diferentes resultados/valores en una lista. Por ejemplo, la función `tabla_del(n)` que se muestra a continuación hace esto:

```
>>> def tabla_del(n):
...     resultados = []
...     for i in range(11):
...         resultados.append(n * i)
...     return resultados
...
>>> res = tabla_del(3)
>>> res
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

En Python una función siempre devuelve un valor

Python, a diferencia de otros lenguajes de programación, no tiene procedimientos. Un procedimiento sería como una función pero que no devuelve ningún valor.

¿Por qué no tiene procedimientos si hemos vistos ejemplos de funciones que no retornan ningún valor? Porque Python, internamente, devuelve por defecto el valor `None` cuando en una función no aparece la sentencia `return` o esta no devuelve nada.

```
>>> def saludo(nombre):
...     print(f'Hola {nombre}')
...
>>> print(saludo('j2logo'))
Hola j2logo
None
```

Como puedes ver en el ejemplo anterior, el `print()` que envuelve a la función `saludo()` muestra `None`.

Parámetros

Paso de parámetros

Tal y como te he indicado, una función puede definir, opcionalmente, una secuencia de parámetros. Los parámetros permiten pasar valores a la función cuando se invoca, conocidos como argumentos. ¿Cómo se asignan en Python los argumentos a los parámetros? Si se modifica el valor de un parámetro, ¿se ve dicho cambio reflejado fuera de la función?

Antes de contestar a estas dos preguntas, vamos a repasar los conceptos de programación conocidos como *paso de variables por valor* y *paso de variables por referencia*.

- **Paso por valor:** Un lenguaje de programación que utiliza *paso de variables por valor*, lo que realmente hace es copiar el valor de las variables en los respectivos parámetros. Cualquier modificación en el parámetro no afecta a la variable externa correspondiente, puesto que ambos referencian a elementos diferentes.
- **Paso por referencia:** En un lenguaje de programación que utiliza *paso de variables por referencia*, los parámetros apuntan exactamente a la misma dirección de memoria que la variable original. Esto implica que, si dentro de la función se modifica el valor de un parámetro, también se modifica en la variable original. En función del lenguaje, el paso por referencia se implementa de formas distintas.

Muchos lenguajes de programación usan a la vez paso por valor y por referencia, según el tipo de la variable. Por ejemplo, paso por valor para los tipos simples: *entero*, *float*, ... y paso por referencia para los objetos. Otros, incluso, permiten al programador que este especifique cómo realizar el paso.

Sin embargo, en Python todo es un objeto y las variables son nombres que identifican a los objetos. Ya vimos esto

en el Tema 3, *Variables*. Entonces, ¿cómo se pasan los argumentos en Python, por valor o por referencia? Lo que ocurre en Python realmente es que **se pasa por valor la referencia del objeto**. ¿Qué implicaciones tiene esto? Básicamente que, si el tipo que se pasa como argumento es inmutable, cualquier modificación en el valor del parámetro no afectará a la variable externa, pero si es mutable (como una lista o diccionario), sí se verá afectado por las modificaciones. Así que, ¡¡¡cuidado!!! Para complementar esta explicación, no te pierdas el tema siguiente.

A continuación, veremos los diferentes tipos de parámetros que se pueden definir en una función.

Valores por defecto de los argumentos

Ya hemos visto que la manera más simple de llamar a una función es pasando un argumento por cada parámetro que define la función. Los argumentos se asignan a los parámetros en el mismo orden en que estos se definen. Estos parámetros son obligatorios y si no se especifica un argumento por cada parámetro en la llamada a la función, el intérprete lanzará un error.

```
>>> def saludo(nombre, rep):
...     print(f'{"¡" * rep}Hola {nombre}{"!" * rep}')
...
>>> saludo('J2logo', 1)
¡Hola J2logo!
>>> saludo('J2logo', 4)
¡¡¡¡Hola J2logo!!!!
```

No obstante, Python permite especificar que un argumento (o varios) sea opcional. Para ello, hay que asignar un valor por defecto en la definición del parámetro:

```
>>> def saludo(nombre, rep=1):
...     print(f'{"¡" * rep}Hola {nombre}{"!" * rep}')
```

La función anterior se puede llamar con un único parámetro o con dos:

```
>>> saludo('J2logo')
¡Hola J2logo!
>>> saludo('J2logo', 3)
¡¡¡Hola J2logo!!!
```

IMPORTANTE: Los parámetros con valores por defecto deben ir siempre después de los parámetros obligatorios.

Argumentos de palabra clave

Según te decía en el apartado anterior, los argumentos se asignan a los parámetros en el mismo orden en que estos se definen en la función.

Sin embargo, en Python existe un modo diferente de indicar los argumentos y es utilizando una palabra clave. Esta palabra clave se corresponde con el nombre del parámetro al que se asigna el argumento. Esto permite pasar los argumentos en cualquier orden.

A continuación, puedes ver un ejemplo:

```
>>> def una_funcion(a, b, c='c', d='d'):
...     print(f'a: {a}')
...     print(f'b: {b}')
...     print(f'c: {c}')
...     print(f'd: {d}')
...
>>> una_funcion(1, c=3, b=2)
a: 1
b: 2
c: 3
```

Cuando se utiliza este modo de pasar los argumentos, se deben cumplir las siguientes reglas:

- Los argumentos con palabra clave deben ir siempre después de los argumentos posicionales.
- No se puede indicar una palabra clave que no se corresponda con el nombre de un parámetro.
- No se puede especificar un argumento dos veces. Por ejemplo, la siguiente llamada a la función daría un error: `una_funcion(3, a=5, b=8)`.

Argumentos indeterminados

En Python podemos usar dos parámetros especiales en la definición de la función que nos permiten pasar un

número indeterminado de argumentos desconocidos. Estos parámetros son `*args` y `**kwargs`.

En realidad, el nombre de estos parámetros puede ser cualquiera. Se usa `args` y `kwargs` por convención. Lo que sí es obligatorio es el uso de los asteriscos y que `*args` siempre debe aparecer antes de `**kwargs`. De hecho, `**kwargs`, si aparece, es el último parámetro de la función. Además, no es necesario que se definan a la vez, pueden aparecer por separado.

Ahora bien, ¿qué significan estos parámetros?

Cuando aparece el parámetro `*args`, este **recibe una tupla con todos los argumentos posicionales tras los argumentos formales** (los que están definidos en la función). Por su parte, `**kwargs` **recibe un diccionario con todos los argumentos con palabra clave excepto aquellos que se corresponden con un parámetro formal**. Después de `*args`, todos los argumentos que se indiquen deben ser argumentos con palabra clave.

Mejor veámoslo con un ejemplo:

```
>>> def una_funcion(a, *args, b, **kwargs):
...     print(a)
...     print(args)
...     print(b)
...     print(kwargs)

>>> una_funcion(1, 2, 3, c=1, d=2, b=5)
1
(2, 3)
5
{'c': 1, 'd': 2}
```

Argumentos posicionales y argumentos con nombre

Como hemos visto, en la llamada a una función se pueden usar al mismo tiempo argumentos posicionales y argumentos con nombre, siempre y cuando estos aparezcan después.

No obstante, es posible indicar qué argumentos deben ser exclusivamente posicionales y cuáles solo argumentos con nombre.

Para ello, la definición de una función debe ser como sigue:

```
def una_funcion(a, b, /, c, *, d, e):
    pass
```

En la función anterior, los parámetros `a` y `b` son posicionales; `d` y `e` se deben pasar como argumentos con nombre y, finalmente, el parámetro `c` se puede pasar como posicional o con nombre.

Es muy raro definir una función de este modo, aunque resulta especialmente útil en este caso.

Imagina la siguiente función:

```
>>> def f(a, **kwargs):
...     pass
```

Como hemos visto, llamar a la función del siguiente modo daría un error porque estaríamos intentado asignar dos argumentos a un mismo parámetro:

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

No obstante, si se redefine la función del siguiente modo, no habría ambigüedad y se podría pasar un argumento con nombre `a` al parámetro `**kwargs`.

```
def f(a, /, **kwargs):
...     pass
...
f(1, a=2)
```

Funciones lambda

Para finalizar este tema te voy a explicar un nuevo tipo de funciones conocidas como *funciones lambda*. Estas funciones son también llamadas *funciones anónimas* porque se definen sin un nombre.

Como hemos visto al comienzo del tema, para definir una función se utiliza la palabra reservada `def`. Sin embargo, una función anónima se define con la palabra reservada `lambda`.

La sintaxis para definir una función lambda es la siguiente:

```
lambda parámetros: expresión
```

A continuación, te muestro las características principales de este tipo de funciones:

- Son funciones que pueden definir cualquier número de parámetros, pero una única expresión. Esta expresión es evaluada y devuelta.
- Se pueden usar en cualquier lugar en el que una función sea requerida.
- Estas funciones están restringidas al uso de una sola expresión.
- Se suelen usar en combinación con otras funciones, generalmente como argumentos de otra función.

Ejemplo:

```
>>> cuadrado = lambda x: x ** 2
>>> cuadrado(4)
16
```

En el ejemplo anterior, `x` es el parámetro y `x ** 2` la expresión que se evalúa y se devuelve.

Como ves, la función no tiene nombre y toda la definición devuelve una función que se asigna al identificador `cuadrado`.

No te pierdas los videotutoriales correspondientes a este tema ya que en ellos se explican otro tipo de funciones y usos avanzados de las mismas.