Funciones (`def`)

Una función es un bloque de código que tiene asociado un nombre, de manera que cada vez que se quiera ejecutar el bloque de código basta con invocar el nombre de la función.

Para declarar una función se utiliza la siguiente sintaxis:

```
def <nombre-funcion> (<parámetros>):
    bloque código
    return <objeto>
```

```
>>> def bienvenida():
...     print('¡Bienvenido a Python!')
...     return
...
>>> type(bienvenida)
<class 'function'>
>>> bienvenida()
¡Bienvenido a Python!
```

Parámetros y argumentos de una función

Una función puede recibir valores cuando se invoca a través de unas variables conocidas como *parámetros* que se definen entre paréntesis en la declaración de la función. En el cuerpo de la función se pueden usar estos parámetros como si fuesen variables.

Los valores que se pasan a la función en una llamada o invocación concreta de ella se conocen como *argumentos* y se asocian a los parámetros de la declaración de la función.

```
>>> def bienvenida(nombre):
...     print('¡Bienvenido a Python', nombre + '!')
...     return
...
>>> bienvenida('Alf')
¡Bienvenido a Python Alf!
```

Paso de argumentos a una función

Los argumentos se pueden pasar de dos formas:

- **Argumentos posicionales**: Se asocian a los parámetros de la función en el mismo orden que aparecen en la definición de la función.
- **Argumentos nominales**: Se indica explícitamente el nombre del parámetro al que se asocia un argumento de la forma `parametro = argumento`.

```
>>> def bienvenida(nombre, apellido):
...     print('¡Bienvenido a Python', nombre, apellido + '!')
...     return
...
>>> bienvenida('Alfredo', 'Sánchez)
```

```
¡Bienvenido a Python Alfredo Sánchez!
>>> bienvenida(apellido = 'Sánchez', nombre = 'Alfredo')
¡Bienvenido a Python Alfredo Sánchez!
```

Retorno de una función

Una función puede devolver un objeto de cualquier tipo tras su invocación. Para ello el objeto a devolver debe escribirse detrás de la palabra reservada `return`. Si no se indica ningún objeto, la función no devolverá nada.

```
>>> def area_triangulo(base, altura):
...     return base * altura / 2
...
>>> area_triangulo(2, 3)
3
>>> area_triangulo(4, 5)
10
```

Una función puede devolver más de un objeto separándolos por comas tras la palabra reservada `return`. En tal caso, la función agrupará los objetos en una tupla y devolverá la tupla.

Argumentos por defecto

En la definición de una función se puede asignar a cada parámetro un argumento por defecto, de manera que si se invoca la función sin proporcionar ningún argumento para ese parámetro, se utiliza el argumento por defecto.

```
>>> def bienvenida(nombre, lenguaje = 'Python'):
...     print('¡Bienvenido a', lenguaje, nombre + '!')
...     return
...
>>> bienvenida('Alf')
¡Bienvenido a Python Alf!
>>> bienvenida('Alf', 'Java')
¡Bienvenido a Java Alf!
```

Los parámetros con un argumento por defecto deben indicarse después de los parámetros sin argumentos por defectos. De lo contrario se produce un error.

Pasar un número indeterminado de argumentos

Por último, es posible pasar un número variable de argumentos a un parámetro. Esto se puede hacer de dos formas:

- `*parametro`: Se antepone un asterisco al nombre del parámetro y en la invocación de la función se pasa el número variable de argumentos separados por comas. Los argumentos se guardan en una lista que se asocia al parámetro.

```
>>> def menu(*platos):
```

```
...     print('Hoy tenemos: ', end='')
...     for plato in platos:
...         print(plato, end=', ')
...     return
...
>>> menu('pasta', 'pizza', 'ensalada')
Hoy tenemos: pasta, pizza, ensalada,
```

- `**parametro`: Se anteponen dos asteriscos al nombre del parámetro y en la invocación de la función se pasa el número variable de argumentos por pares `nombre = valor`, separados por comas. Los argumentos se guardan en un diccionario que se asocia al parámetro.

```
>>> def contacto(**info):
...     print('Datos del contacto)
...     for clave, valor in info.items():
...         print(clave, ":", valor)
...     return
...
>>> contacto(Nombre = "Alf", Email = "asalber@email.ar")
Datos del contacto
Nombre : Alf
Email : asalber@email.ar
>>> contacto(Nombre = "Alf", Email = "asalber@email.ar", Dirección = "Madrid")
Datos del contacto
Nombre : Alf
Email : asalber@email.ar
Dirección : Madrid
```

Ámbito de los parámetros y variables de una función

Los parámetros y las variables declaradas dentro de una función son de **ámbito local**, mientras que las definidas fuera de ella son de ámbito **ámbito global**.

Tanto los parámetros como las variables del ámbito local de una función sólo están accesibles durante la ejecución de la función, es decir, cuando termina la ejecución de la función estas variables desaparecen y no son accesibles desde fuera de la función.

```
>>> def bienvenida(nombre):
...     lenguaje = 'Python'
...     print('¡Bienvenido a', lenguaje, nombre + '!')
...     return
...
>>> bienvenida('Alf')
¡Bienvenido a Python Alf!
>>> lenguaje
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'lenguaje' is not defined
```

Si en el ámbito local de una función existe una variable que también existe en el ámbito global, durante la ejecución de la función la variable global queda eclipsada por la variable local y no es accesible hasta que finaliza la ejecución de la función.

```
>>> lenguaje = 'Java'
>>> def bienvenida():
...     lenguaje = 'Python'
...     print('¡Bienvenido a', lenguaje + '!')
...     return
...
>>> bienvenida()
¡Bienvenido a Python!
>>> print(lenguaje)
Java
```

Paso de argumentos por asignación

En Python los argumentos se pasan a una función por asignación, es decir, se asignan a los parámetros de la función como si fuesen variables locales. De este modo, cuando los argumentos son objetos mutables (listas, diccionarios, etc.) se pasa al parámetro una referencia al objeto, de manera que cualquier cambio que se haga dentro de la función mediante el parámetro asociado afectará al objeto original.

```
>>> primer_curso = ['Matemáticas', 'Física']
>>> def añade_asignatura(curso, asignatura):
...     curso.append(asignatura)
...     return
...
>>> añade_asignatura(primer_curso, 'Química')
>>> print(primer_curso)
['Matemáticas', 'Física', 'Química']
```

Las funciones son objetos

En Python las funciones son objetos como el resto de tipos de datos, de manera que es posible asignar una función a una variable y luego utilizar la variable para hacer la llamada a la función.

```
>>> def saludo(nombre):
...     print("Hola", nombre)
...     return
...
>>> bienvenida = saludo
>>> bienvenida("Alf")
Hola Alf
```

Esto permite, por tanto, pasar funciones como argumentos en la llamada a una función y que una función pueda devolver otras funciones.

```
>>> def impuesto(porcentaje):
...     def aplicar(base):
...             return base * porcentaje / 100
```

```
...     return aplicar
...
>>> iva = impuesto(21)
>>> iva(1000)
210.0
```

Funciones recursivas

Una función recursiva es una función que en su cuerpo contiene una llama a si misma.

La recursión es una práctica común en la mayoría de los lenguajes de programación ya que permite resolver las tareas recursivas de manera más natural.

Para garantizar el final de una función recursiva, las sucesivas llamadas tienen que reducir el grado de complejidad del problema, hasta que este pueda resolverse directamente sin necesidad de volver a llamar a la función.

```
>>> def factorial(n):
...     if n == 0:
...         return 1
...     else:
...         return n * factorial(n-1)
...
>>> f(5)
120
```

Funciones recursivas múltiples

Una función recursiva puede invocarse a si misma tantas veces como quiera en su cuerpo.

```
>>> def fibonacci(n):
...     if n <= 1:
...         return n
...     else:
...         return fibonacci(n - 1) + fibonacci(n - 2)
...
>>> fibonacci(6)
8
```

Los riesgos de la recursión

Aunque la recursión permite resolver las tareas recursivas de forma más natural, hay que tener cuidado con ella porque suele consumir bastante memoria, ya que cada llamada a la función crea un nuevo ámbito local con las variables y los parámetros de la función.

En muchos casos es más eficiente resolver la tarea recursiva de forma iterativa usando bucles.

```
>>> def fibonacci(n):
...     a, b = 0, 1
...     for i in range(n):
...         a, b = b, a + b
...     return a
```

```
...
>>> fibonacci(6)
8
```

Documentación de funciones

Una práctica muy recomendable cuando se define una función es describir lo que la función hace en un comentario.

En Python esto se hace con un **docstring** que es un tipo de comentario especial se hace en la línea siguiente al encabezado de la función entre tres comillas simples `'''` o dobles `"""`.

Después se puede acceder a la documentación de la función con la función `help(<nombre-función>)`.

```
>>> def area_triangulo(base, altura):
... """Función que calcula el área de un triángulo.
...
... Parámetros:
...     - base: Un número real con la base del triángulo.
...     - altura: Un número real con la altura del triángulo.
... Salida:
...     Un número real con el área del triángulo de base y altura
especificadas.
... """
...     return base * altura / 2
...
>>> help(area_triangulo)
area_triangulo(base, altura)
    Función que calcula el área de un triángulo.

    Parámetros:
        - base: Un número real con la base del triángulo.
        - altura: Un número real con la altura del triángulo.
    Salida:
        Un número real con el área del triángulo de base y altura
especificadas.
```