Baldomero Sánchez

1. Funciones

Unidad 5

funciones

Una función es una sección de un programa o una parte de código que calcula un valor o realiza alguna tarea de manera independiente al resto del programa.

```
def mi_funcion(arg1, arg2):
    print(arg1)
    print(arg2)
```

Una función tiene tres componentes importantes:

- Los parámetros, que son los valores que recibe la función como entrada.
- El código de la función, que son las operaciones que hace la función.
- El resultado (o valor de retorno), que es el valor final que entrega la función.

En esencia, una función es un mini programa.

funciones

La sintaxis utilizada para definir funciones en Python es la siguiente:

```
def <nombre_funcion>([<parámetros>]):
    <sentencia(s)>
```

```
def mi_funcion(arg1, arg2):
    print(arg1)
    print(arg2)
```

El último elemento, `<sentencia(s)>`, se denomina cuerpo de la función.

El cuerpo es el bloque de sentencias en Python que se ejecutará cuando se llame a la función.

El cuerpo de una función de Python se define por la sangría.

funciones

La sintaxis utilizada para invocar una función y que se ejecute su código es la siguiente:

```
<nombre_funcion>([<argumentos>])
```

Los <argumentos> son los valores que se pasan a la función.

Se corresponden con los <parámetros> en la definición de la función en Python.

Se pueden definir funciones que no reciban ningún argumento, pero los paréntesis siguen siendo necesarios. Tanto una definición de función como una llamada a una función deben incluir siempre paréntesis, incluso si están vacíos.

```
def mi_funcion(arg1, arg2):
    print(arg1)
    print(arg2)


mi_funcion("Hola", "Juan")
```

```
def funcion_saludo():
    print("Hola!!")

funcion_saludo()
```

funciones

La forma más sencilla de pasar argumentos a una función en Python es con argumentos posicionales (también llamados argumentos requeridos).

En la definición de la función, debe especificarse una lista de parámetros separada por comas dentro de los paréntesis.

El número de argumentos que le proporcionamos en la invocación de la función debe respetar el orden y el número de parámetros definidos.

Otra forma de invocar una función en Python es indicando los argumentos de la forma `<palabra clave>=<valor>`. En ese caso, cada `<palabra clave>` debe coincidir con un parámetro en la definición de la función.

```
def mi_funcion(arg1, arg2):
    print(arg1)
    print(arg2)


mi_funcion("Hola", "Juan")
```

```
mi_funcion(arg2="Hola", arg1="Juan")
```

Funciones

•El principio de **reusabilidad**, que nos dice que si por ejemplo tenemos un fragmento de código usado en muchos sitios, la mejor solución sería pasarlo a una función.
• Y el principio de **modularidad**, que defiende que en vez de escribir largos trozos de código.

```
def nombre_funcion(argumentos):
        código return retorno
```

Pasando argumentos de entrada

```
def di_hola(nombre):
        print("Hola", nombre)
di_hola("Juan")
 # Hola Juan
```

funciones

Utilizar argumentos de palabra clave nos exime de tener que respetar el orden de los parámetros definidos en la función, pero el número de argumentos debe seguir respetándose.

Podemos combinar argumentos posicionales y de palabra clave en la misma llamada a una función. En estos casos, los argumentos posicionales deben indicarse primero.

```
def mi_funcion(arg1, arg2, arg3):
    print(arg1)
    print(arg2)
    print(arg3)


mi_funcion("Hola", arg3="Garcia", arg2="Juan")
```

Funciones

Argumentos por posición

Los argumentos por posición o posicionales son la forma más básica e intuitiva de pasar parámetros. Si tenemos una función resta() que acepta dos parámetros, se puede llamar como se muestra a continuación.

```
def resta(a, b):
    return a-b
resta(5, 3) # 2
```

Si se pasan menos parámetros, da un error

No es posible usar mas argumentos de los tiene la función definidos, ya que no sabría que hacer con ellos. Por lo tanto si lo intentamos, Python nos dirá que toma 2 posicionales y estamos pasando 3, lo que no es posible.

```
#TypeError: resta() takes 2 positional arguments but 3 were given
#resta(5,4,3) # Error
```

Funciones – argumentos por nombre

Otra forma de llamar a una función, es usando el nombre del argumento con = y su valor. El siguiente código hace lo mismo que el código anterior, con la diferencia de que los argumentos no son posicionales.

```
resta(a=3, b=5) # -2
```

Al indicar en la llamada a la función el nombre de la variable y el valor, el orden ya no importa, y se podría llamar de la siguiente forma.

```
resta(b=5, a=3) # -2
```

Como es de esperar, si indicamos un argumento que no ha sido definido como parámetro de entrada, tendremos un error

```
#resta() got an unexpected keyword argument 'c'
#resta(b=5, c=3) # Error!
```

Funciones – argumentos por nombre

Otra forma de llamar a una función, es usando el nombre del argumento con = y su valor. El siguiente código hace lo mismo que el código anterior, con la diferencia de que los argumentos no son posicionales.

```
resta(a=3, b=5) # -2
```

Al indicar en la llamada a la función el nombre de la variable y el valor, el orden ya no importa, y se podría llamar de la siguiente forma.

```
resta(b=5, a=3) # -2
```

Como es de esperar, si indicamos un argumento que no ha sido definido como parámetro de entrada, tendremos un error

```
#resta() got an unexpected keyword argument 'c'
#resta(b=5, c=3) # Error!
```

Funciones – argumentos por defecto

Se pueden asignar un valor **por defecto** a la función

```
def suma(a, b, c=0):
        return  a+b+c
suma(43,15,3)
# 61
```

Si se llama a la función y solo se pasan 2 argumentos, el tercero asume el valor por defecto establecido.

```
suma(3,8)
# 11
```

Si solo se invoca a la función y no se pasa ningún parámetro

```
def suma(a=3, b=5, c=0):
        return a+b+c
suma()
# 8
```

```
suma(1) # 6
 suma(4,5) # 9
suma(5,3,2) # 10
```

Funciones – argumentos de longitud variable

a) Una primera forma sería con una lista.

.

```
def suma(numeros):
        total = 0
        for n in numeros:
                total += n
        return total
suma([1,3,5,4]) # 13
```

Funciones – argumentos de longitud variable

b) Python tiene una herramienta muy potente.

Si declaramos un argumento precedido con *, esto hará que el argumento que se pase **sea empaquetado en una tupla** de manera automática.

```
def suma(*numeros):
        print(type(numeros))
        # <class 'tuple'>
        total = 0
        for n in numeros:
                total += n
        return total
suma(1, 3, 5, 4) # 13
```

```
suma(1) # 6
 suma(4,5) # 9
suma(5,3,78,4,23,2) #
115
```

Funciones – argumentos de longitud variable

c) Usando doble ** es posible también tener como parámetro de entrada una lista de elementos almacenados en forma de clave y valor.

En este caso podemos iterar los valores haciendo uso de items().

```
def suma(**kwargs):
        suma = 0;
        for key, value in kwargs.items():
                print(key, value)
                suma += value
        return suma
suma(a=5, b=20, c=23) # 48
```

d) Partiendo del punto anterior, podemos pasar un diccionario como parámetro de entrada.

```
def suma(**kwargs):
        suma = 0
        for key, value in wargs.items():
                print(key, value)
                suma += value
        return suma
di = {'a': 10, 'b':20}
suma(**di) # 30
```

Funciones- Sentencia return

Si indicamos junto con la palabra `return` un valor, dicho valor es devuelto, cuando acabe de ejecutarse el código de la función y llegué a la palabra `return`, en la misma línea en la que se produce la llamada a la función.

Si no se indica ningún valor junto la palabra `return` se devuelve la palabra reservada None.

```
def mi_funcion():
    return “Hola Juan”


mensaje=mi_función()
print(mensaje)
```

```
Hola Juan
```

Funciones- Sentencia return

El uso de la sentencia return permite realizar dos cosas:

a) Salir de la función y transferir la ejecución de vuelta a donde se realizó la llamada.
b) Devolver uno o varios parámetros, fruto de la ejecución de la función.

En lo relativo a lo primero, una vez se llama a return se para la ejecución de la función y se vuelve o retorna al punto donde fue llamada

```
def mi_funcion():
    return "Hola Juan"


mensaje=mi_función()
print(mensaje)
```

```
Hola Juan
```

```
def suma_y_media(a, b, c):
        suma = a+b+c
        media = suma/3
        return suma, media
suma, media = suma_y_media(9, 6, 3)
print(suma) # 18
print(media) # 6.0
```

funciones

Una función puede devolver varios valores a la vez especificándolos a continuación del `return` y separándolos por comas.

Para recogerlos podemos usar variables separadas por comas.

```
def mi_funcion():
    return "Hola", "Juan", "Adios"


mensaje1, mensaje2, mensaje3=mi_función()
print(mensaje1)
```

```
Hola
```

Funciones- Documentación

Cuando la primera declaración en el cuerpo de una función en Python es un comentario de la forma cadenas de caracteres multilínea de triple comilla doble (""""), se le conoce como ***docstring*** de la función.

- Los *docstring* se utilizan para proporcionar la documentación de una función.
- Puede contener el propósito de la función, los argumentos que toma, información sobre los valores de retorno, o cualquier otra información que se considere útil.

Existe un PEP (*Python Enhancement Proporsal*) que establece el estilo con el que deben definirse los *docstrings*: https://www.python.org/dev/peps/pep-0257/

```
def mi_funcion():
    """Esto es un docstring de la
función."""
    return “Hola”, “Juan”, “Adios”
```

Funciones- Documentación

Cualquiera que tenga nuestra función, podrá llamar a la función help() y obtener la ayuda de como debe ser usada.

help(mi_funcion_suma)

Otra forma de acceder a la documentación es la siguiente.

print(mi_funcion_suma.__doc__)

```
def mi_funcion():
    """Esto es un docstring de la
función."""
    return “Hola”, “Juan”, “Adios”
```

Anotaciones en Funciones

Existe una funcionalidad relativamente reciente en Python llamada function annotation o anotaciones en funciones.
Esta funcionalidad nos permite añadir metadatos a las funciones, indicando:

- Los tipos esperados tanto de entrada como de salida.

```
def multiplica_por_3(numero: int) -> int:
        return numero*3
multiplica_por_3(6) # 18
```

Las anotaciones son muy útiles de cara a la documentación del código, pero no imponen ninguna norma sobre los tipos. Esto significa que se puede llamar a la función con un parámetro que no sea int, y no obtendremos ningún error.

```
multiplica_por_3("Cadena") #
'CadenaCadenaCadena'
```

ámbito de las variables

El ámbito de una variable (*scope*) es la zona del programa en la que se define la variable.

Las variables en Python son locales por defecto. Por lo que las variables definidas y utilizadas en el bloque de código de una función, sólo tienen existencia dentro de la misma, y no interfieren con otras variables del resto del código.

En caso de que sea conveniente o necesario, una variable local puede convertirse en una variable global declarándola explícitamente como tal con la sentencia `global`.

```
# definimos una funcion llamada local
def local():
 m = 7
 print(m)

m = 5
print(m)

# llamanos a funcion local
local()
```

```
global numero
```

built-in functions

El intérprete de Python tiene un conjunto de funciones útiles que están siempre disponibles y que forman parte de la librería "estándar" de Python (*Built-in functions*).

Están disponibles a través del siguiente enlace a la documentación de Python:

https://docs.python.org/3/library/functions.html

Built-in Functions				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

built-in functions

La función `help()` es muy útil porque da acceso a la documentación de Python.

Por si sola muestra la ayuda, pero se le puede pasar algo determinado como argumento para que busque y muestre la ayuda referente a ese tema concreto.

Otras funciones Built-in que ya hemos visto son: `len()`, `float()`, `int()`, `str()`, `type()`.

```
help(float)
```

```
Help on class float in module builtins:

class float(object)
 |  float(x=0, /)
 |
 |  Convert a string or number to a floating point number, if possible.
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __bool__(self, /)
 |      self != 0
```

Bibliografía y recursos online

- https://www.mclibre.org/consultar/python/index.html
- https://docs.python.org/es/3/reference
- https://www.python.org/dev/peps/pep-0257/
- https://docs.python.org/3/library/functions.html