Funciones de Python

Una función es un bloque de código que solo se ejecuta cuando se llama.

Puede pasar datos, conocidos como parámetros, a una función.

Una función puede devolver datos como resultado.

Crear una función

En Python, una función se define usando la palabra clave def:

Example

def my\_function():  
  print("Hello from a function")

Llamar a una función

Para llamar a una función, use el nombre de la función seguido de paréntesis:

Example

def my\_function():  
  print("Hello from a function")  
  
**my\_function()**

Argumentos

La información se puede pasar a funciones como argumentos.

Los argumentos se especifican después del nombre de la función, entre paréntesis. Puede agregar tantos argumentos como desee, solo sepárelos con una coma.

El siguiente ejemplo tiene una función con un argumento (fname). Cuando se llama a la función, pasamos un nombre, que se usa dentro de la función para imprimir el nombre completo:

Example

def my\_function(**fname**):  
  print(fname + " Refsnes")  
  
my\_function(**"Emil"**)  
my\_function(**"Tobias"**)  
my\_function(**"Linus"**)

Los argumentos a menudo se acortan a *args* en las documentaciones de Python.

¿Parámetros o argumentos?

Los términos parámetro y argumento se pueden usar para lo mismo: información que se pasa a una función.

Desde la perspectiva de una función:

Un parámetro es la variable que aparece entre paréntesis en la definición de la función.

Un argumento es el valor que se envía a la función cuando se llama.

Número de argumentos

De forma predeterminada, se debe llamar a una función con el número correcto de argumentos. Lo que significa que, si su función espera 2 argumentos, debe llamar a la función con 2 argumentos, ni más ni menos.

### Example

Esta función espera 2 argumentos y obtiene 2 argumentos:

def my\_function(fname, lname):  
  print(fname + " " + lname)  
  
my\_function("Emil", "Refsnes")

Si intenta llamar a la función con 1 o 3 argumentos, obtendrá un error:

Example

Esta función espera 2 argumentos, pero solo obtiene 1:

def my\_function(fname, lname):  
  print(fname + " " + lname)  
  
my\_function("Emil")

Argumentos arbitrarios, \*args

Si no sabe cuántos argumentos se pasarán a su función, agregue un \* antes del nombre del parámetro en la definición de la función.

De esta forma, la función recibirá una tupla de argumentos y podrá acceder a los elementos en consecuencia:

### Example

Si se desconoce el número de argumentos, agregue un \* antes del nombre del parámetro:

def my\_function(\*kids):  
  print("The youngest child is " + kids[2])  
  
my\_function("Emil", "Tobias", "Linus")

*Los argumentos arbitrarios a menudo se acortan a \*args en la documentación de Python.*

## Argumentos de palabras clave (Keyword Arguments)

También puede enviar argumentos con la sintaxis clave = valor.

De esta forma no importa el orden de los argumentos.

### Example

def my\_function(child3, child2, child1):  
  print("The youngest child is " + child3)  
  
my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

La frase *Keyword Arguments* a menudo se abrevia a *kwargs* en las documentaciones de Python.

Argumentos de palabras clave arbitrarias, \*\* kwargs

Si no sabe cuántos argumentos de palabras clave se pasarán a su función, agregue dos asteriscos: \*\* antes del nombre del parámetro en la definición de la función.

De esta forma, la función recibirá un diccionario de argumentos y podrá acceder a los elementos en consecuencia:

### Example

Si se desconoce el número de argumentos de palabras clave, agregue un doble \*\* antes del nombre del parámetro:

def my\_function(\*\*kid):  
  print("His last name is " + kid["lname"])  
  
my\_function(fname = "Tobias", lname = "Refsnes")

*Los argumentos arbitrarios de Keyword a menudo se acortan a \*\*kwargs en las documentaciones de Python.*

Valor predeterminado de parámetro

El siguiente ejemplo muestra cómo utilizar un valor de parámetro predeterminado.

Si llamamos a la función sin argumento, usa el valor predeterminado:

Example

def my\_function(**country = "Norway"**):  
  print("I am from " + country)  
  
my\_function("Sweden")  
my\_function("India")  
my\_function()  
my\_function("Brazil")

Pasar una lista como argumento

Puede enviar cualquier tipo de datos de argumento a una función (cadena, número, lista, diccionario, etc.), y se tratará como el mismo tipo de datos dentro de la función.

P.ej. si envía una Lista como argumento, seguirá siendo una Lista cuando llegue a la función:

### Example

def my\_function(food):  
  for x in food:  
    print(x)  
  
fruits = ["apple", "banana", "cherry"]  
  
my\_function(fruits)

Valores devueltos

Para permitir que una función devuelva un valor, use la declaración de return:

Example

def my\_function(x):  
  **return 5 \* x**  
print(my\_function(3))  
print(my\_function(5))  
print(my\_function(9))

La declaración de pass

Las definiciones de función no pueden estar vacías, pero si por alguna razón tiene una definición de función sin contenido, coloque la instrucción pass para evitar errores.

### Example

def myfunction():  
  pass

# tener una definición de función vacía como esta, generaría un error # sin la declaración de paso

Recursividad

Python también acepta la recursividad de funciones, lo que significa que una función definida puede llamarse a sí misma.

La recursividad es un concepto matemático y de programación común. Significa que una función se llama a sí misma. Esto tiene la ventaja de significar que puede recorrer los datos para llegar a un resultado.

El desarrollador debe tener mucho cuidado con la recursividad, ya que puede ser bastante fácil escribir una función que nunca termina, o una que usa cantidades excesivas de memoria o potencia del procesador. Sin embargo, cuando se escribe correctamente, la recursividad puede ser un enfoque de programación muy eficiente y matemáticamente elegante.

En este ejemplo, tri\_recursion() es una función que hemos definido para llamarse a sí misma ("recurse"). Usamos la variable k como dato, que disminuye (-1) cada vez que recurrimos. La recursividad termina cuando la condición no es mayor que 0 (es decir, cuando es 0).

Para un desarrollador nuevo, puede tomar algún tiempo averiguar cómo funciona exactamente, la mejor manera de averiguarlo es probándolo y modificándolo.

### Example

Ejemplo de recursividad

def tri\_recursion(k):  
  if(k > 0):  
    result = k + tri\_recursion(k - 1)  
    print(result)  
  else:  
    result = 0  
  return result  
  
print("\n\nRecursion Example Results")  
tri\_recursion(6)

# Python Lambda

Una función lambda es una pequeña función anónima.

Una función lambda puede tomar cualquier número de argumentos, pero solo puede tener una expresión.

Sintaxis

lambda arguments : expression

Se ejecuta la expresión y se devuelve el resultado:

### Example

Agregue 10 al argumento a y devuelva el resultado:

x = lambda a : a + 10  
print(x(5))

Las funciones Lambda pueden tomar cualquier número de argumentos:

Example

Multiplica el argumento a con el argumento b y devuelve el resultado:

x = lambda a, b : a \* b  
print(x(5, 6))

### Example

Resuma los argumentos a, b y c y devuelva el resultado:

x = lambda a, b, c : a + b + c  
print(x(5, 6, 2))

¿Por qué utilizar las funciones Lambda?

El poder de lambda se muestra mejor cuando los usa como una función anónima dentro de otra función.

Digamos que tiene una definición de función que toma un argumento, y ese argumento se multiplicará por un número desconocido:

def myfunc(n):  
  return lambda a : a \* n

Use esa definición de función para hacer una función que siempre duplique el número que envía:

Example

def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
  
print(mydoubler(11))

O use la misma definición de función para hacer una función que siempre triplique el número que envía:

Example

def myfunc(n):  
  return lambda a : a \* n  
  
mytripler = myfunc(3)  
  
print(mytripler(11))

O use la misma definición de función para hacer ambas funciones, en el mismo programa:

### Example

def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
mytripler = myfunc(3)  
  
print(mydoubler(11))  
print(mytripler(11))

Utilice funciones lambda cuando se requiera una función anónima durante un período corto de tiempo.