Python Functions

June 4, 2024

0.1 Python Functions

Dr. Labeed Al-Saad

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

0.2 Creating a Function

In Python a function is defined using the def keyword:

Example:

[1]:
```
def my_function():
  print("Hello from a function")
```

0.3 Calling a Function

To call a function, use the function name followed by parenthesis:

Example:

[2]:
```
def my_function():
  print("Hello from a function")

my_function()
```

```
Hello from a function
```

0.4 Arguments

Information can be passed into functions as arguments.

**Arguments are often shortened to args in Python documentations.

Arguments are specified after the function name, inside the parentheses. **You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example:

[3]:
```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

0.5 Parameters or Arguments?

The terms parameter and argument can be used for the same thing: information that are passed into a function.

From a function's perspective:

**A parameter is the variable listed inside the parentheses in the function definition.

**An argument is the value that is sent to the function when it is called.

0.6 Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example:

This function expects 2 arguments, and gets 2 arguments:

[4]:
```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

```
Emil Refsnes
```

**If you try to call the function with 1 or 3 arguments, you will get an error:

Example:

This function expects 2 arguments, but gets only 1:

[5]:
```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 4
```

```
      1 def my_function(fname, lname):
      2   print(fname + " " + lname)
----> 4 my_function("Emil")

TypeError: my_function() missing 1 required positional argument: 'lname'
```

0.7 Arbitrary Arguments, *args

**Arbitrary Arguments are often shortened to *args in Python documentations.

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

Example:

If the number of arguments is unknown, add a * before the parameter name:

[6]:
```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

```
The youngest child is Linus
```

0.8 Keyword Arguments

**The phrase Keyword Arguments are often shortened to kwargs in Python documentations.

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

Example:

[7]:
```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

```
The youngest child is Linus
```

0.9 Arbitrary Keyword Arguments, **kwargs

Arbitrary Kword Arguments are often shortened to kwargs in Python documentations.

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

Example:

If the number of keyword arguments is unknown, add a double ** before the parameter name:

[8]:
```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

```
His last name is Refsnes
```

0.10 Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

Example:

[9]:
```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

0.11 Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example:

[10]:
```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

```
apple
banana
cherry
```

0.12 Return Values

To let a function return a value, use the return statement:

Example:

[11]:
```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

```
15
25
45
```

0.13 The pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example:

[12]:
```
def myfunction():
  pass
```

0.14 Positional-Only Arguments

You can specify that a function can have ONLY positional arguments, or ONLY keyword arguments.

To specify that a function can have only positional arguments, add , / after the arguments:

Example:

[13]:
```
def my_function(x, /):
  print(x)

my_function(3)
```

```
3
```

Without the , / you are actually allowed to use keyword arguments even if the function expects positional arguments:

Example:

[14]:
```
def my_function(x):
  print(x)

my_function(x = 3)
```

```
3
```

But when adding the , / you will get an error if you try to send a keyword argument:

Example:

[15]:
```
def my_function(x, /):
  print(x)

my_function(x = 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 4
      1 def my_function(x, /):
      2   print(x)
----> 4 my_function(x = 3)

TypeError: my_function() got some positional-only arguments passed as keyword␣
 ↪arguments: 'x'
```

0.15 Keyword-Only Arguments

To specify that a function can have only keyword arguments, add *, before the arguments:

Example:

[16]:
```
def my_function(*, x):
  print(x)

my_function(x = 3)
```

```
3
```

[ ]:
```
Without the *, you are allowed to use positionale arguments even if the␣
 ↪function expects keyword arguments:

Example:
```

[17]:
```
def my_function(x):
  print(x)

my_function(3)
```

```
3
```

But when adding the *, / you will get an error if you try to send a positional argument:

Example

[18]:
```
def my_function(*, x):
  print(x)
```

```
my_function(3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[18], line 4
      1 def my_function(*, x):
      2   print(x)
----> 4 my_function(3)

TypeError: my_function() takes 0 positional arguments but 1 was given
```

0.16 Combine Positional-Only and Keyword-Only

You can combine the two argument types in the same function.

Any argument before the / , are positional-only, and any argument after the *, are keyword-only.

Example:

[19]:
```
def my_function(a, b, /, *, c, d):
  print(a + b + c + d)

my_function(5, 6, c = 7, d = 8)
```

26

0.17 Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Example:

[20]:
```
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)
```

```
Recursion Example Results
1
3
6
10
15
21
```

[20]: 21