Python Functions

❮ Previous Next ❯

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.

Creating a Function

In Python a function is defined using the `def` keyword:

Example

Get your own Python Server

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

Calling a Function

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

Example

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

my_function()
```

Try it Yourself »

Arguments

Information can be passed into functions as arguments.

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

```
def my_function(fname):
  print(fname + " Refsnes")
```

Try it Yourself »

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

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.

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:

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

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

Try it Yourself »

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:

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

my_function("Emil")
```

Try it Yourself »

Arbitrary Arguments, *args

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.

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

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

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

Try it Yourself »

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

Keyword Arguments

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

This way the order of the arguments does not matter.

Example

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

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

Try it Yourself »

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

Arbitrary Keyword Arguments, **kwargs

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:

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

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

Try it Yourself »

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

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

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

Example

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

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

Try it Yourself »

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

```
def my_function(food):
  for x in food:
    print(x)

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

my_function(fruits)
```

Try it Yourself »

Return Values

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

Example

```
def my_function(x):
  return 5 * x

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

Try it Yourself »

The pass Statement

Example

```
def myfunction():
  pass
```

Try it Yourself »

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

```
def my_function(x, /):
  print(x)

my_function(3)
```

Try it Yourself »

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

Example

```
def my_function(x):
  print(x)

my_function(x = 3)
```

Try it Yourself »

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

Example

```
def my_function(x, /):
  print(x)

my_function(x = 3)
```

Try it Yourself »

Keyword-Only Arguments

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

Example

Try it Yourself »

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

Example

```
def my_function(x):
  print(x)

my_function(3)
```

Try it Yourself »

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

Example

```
def my_function(*, x):
  print(x)

my_function(3)
```

Try it Yourself »

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

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

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

Try it Yourself »

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.

Example

Recursion Example

```
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)
```

Try it Yourself »

Test Yourself With Exercises

Exercise:

Create a function named `my_function`.

```
        :
  print("Hello from a function")
```

Submit Answer »

Start the Exercise

‹ Previous Next ›

COLOR PICKER