IIT Madras
BSc Degree

[Home](#) [Lesson-4.2](#)

# Lesson-4.1

## Functions

### Introduction

In mathematics, a function is an object that accepts one or more inputs and produces one or more outputs. For example, $f(x) = x^2$, is a function that accepts a number and returns the square of that number. Functions in Python play a similar role, but are much more richer than their mathematical counterparts. Let us quickly convert the mathematical function, $f(x) = x^2$, into a Python function:

```python
def f(x):
    y = x ** 2
    return y
```

The code given above is called the **definition** of function `f`. `def` is the keyword used to define functions. `f` is the name of the function. `x` is a parameter of the function. Lines 2 and 3 make up the body of the function and are indented. The body of a function is a collection of statements that describe what the function does. At line-3, the value stored in variable `y` is returned. `return` is the keyword used for this purpose.

If we run the above code, we will not get any output. Functions are not executed unless they are called. The following code demonstrates what a **function call** looks like:

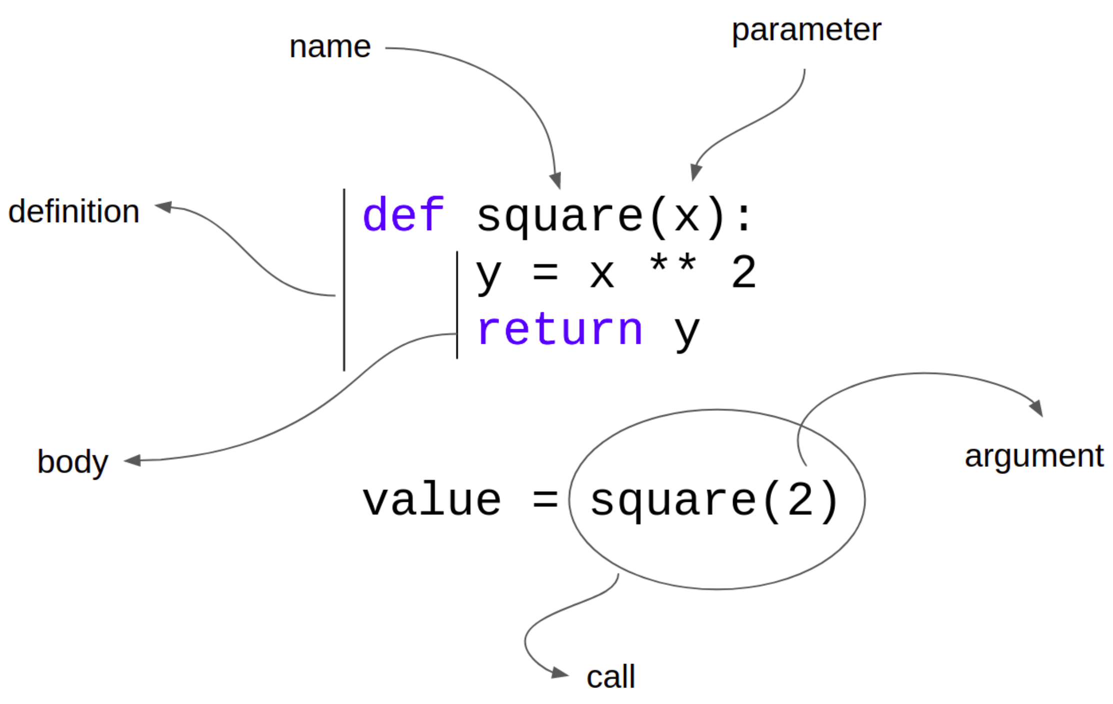
```python
1 def square(x):
2     y = x ** 2
3     return y
4
5 print(square(2))
```

The output is:

```
4
```

`square(2)` is a function call. We use the name of the function, `square`, and pass the number 2 as an argument to it. The `x` in the function definition is called the **parameter**. The value that is passed to the function in the call is called the **argument**. This is a convention that we will follow throughout this lesson.

A visual representation of the terms we have defined so far is given below:



A mental model to understand functions:

- Parameters can be thought of as the function's inputs.
- The body of the function can be pictured as the sequence of steps that transform the input into the output.
- The return statement can be thought of as a means of communicating the output to the rest of the code.

## Examples

We will look at a wide variety of function definitions. The focus will be on the syntactical aspects of function definitions.

- Functions could have multiple parameters:

```
# This function computes the area of a rectangle.
# Length and breadth are the parameters
def area(l, b):
    return l * b
```

- Functions could have no parameters:

```python
def foo():
    return "I don't like arguments visiting me!"
```

- Functions could have no return value:

```python
def foo():
    print("I don't like talking to the outside world!")

foo()
```

When the code given above is executed, we get the following output:

```
I don't like talking to the outside world!
```

Note that we didn't have to type `print(foo())`. We just had to call the function — `foo()` — since it already has the print statement inside it. But what happens if we type `print(foo())`? We get the following output:

```
I don't like talking to the outside world!
None
```

If no explicit return statement is present in a function, `None` is the default value returned by it. When the interpreter comes across the `print(foo())` statement, first the function `foo()` is evaluated. This results in the first line of the output. Since `foo()` has no explicit return statement, it returns `None` by default. That is why the second line in the output is `None`.

- A minimal Python function looks like the one given below:

```python
def foo():
    pass
```

`pass` is a keyword in Python. When the interpreter comes across a `pass` statement, it doesn't perform any computation and moves on to the next line. The reason this is minimal is because it has only those features that are absolutely essential for a function definition to be syntactically valid: function name and at least one statement in the body.

Such functions might seem useless at first sight, but they do have their place in programming. While writing a complex piece of code, a coder may realize that she needs to define a function to perform a specific task. But she may not know the exact details of the implementation or it may not be an urgent requirement. In such a scenario, she will add a minimal function like the one given above in her code and name it appropriately. Implementing this function will become a task on her to-do list and will be taken up as and when the need arises.

- Functions could have multiple return statements, but the moment the first return is executed, control exits from the function:

```python
def foo():
    return 1
    return 2
```

`foo()` will always return 1. Line-3 is redundant. An example of a function having multiple returns that are not redundant:

```
1  def evenOrOdd(n):
2      if n % 2 == 0:
3          return 'even'
4      else:
5          return 'odd'
6
7  print(evenOrOdd(10))
8  print(evenOrOdd(11))
```

The output is:

```
1  even
2  odd
```

When `evenOrOdd` is called with an even number as argument, the return statement in line-3 is executed. When the same function is called with an odd number as argument, the return statement in line-5 is executed.

- Functions could return multiple values:

```
1  # Accept only positive floating point numbers
2  def bound(x):
3      lower = int(x)
4      upper = lower + 1
5      return lower, upper
6
7  y = 7.3
8  l, u = bound(y)
9  print(f'{l} < {y} < {u}')
```

The exact mechanism of what happens here will become clear when we come to the lesson on tuples. In line-8, the first value returned by `bound` is stored in `l` and the second value returned by `bound` is stored in `u`.

- Functions have to be defined before they can be called. The function call cannot come before the definition. For example:

```
1  ##### Alarm! Wrong code snippet! #####
2  print(f(5))
3
4  def f(x):
5      return x ** 2
6  ##### Alarm! Wrong code snippet! #####
```

When the above code is executed, it throws a `NameError`. Why does this happen? The Python interpreter executes the code from top to bottom. At line-2, `f` is a name that the interpreter has never seen before and therefore it throws a `NameError`. Recall that `NameError` occurs when we try to reference a name that the interpreter has not seen before.

- Function calls could be used in expressions:

```python
def square(a):
    return a ** 2

x, y, z = int(input()), int(input()), int(input())
if square(x) + square(y) == square(z):
    print(f'{x}, {y} and {z} form the sides of a right triangle with {z} as the hypotenuse')
```

- Function calls cannot be assigned values:

```python
##### Alarm! Wrong code snippet! #####
def foo():
    return True

foo() = 1
##### Alarm! Wrong code snippet! #####
```

The above code throws a `SyntaxError`.

- Functions can be called from within other functions:

```python
def foo():
    print('I am inside foo')

def bar():
    print('I am inside bar')
    print('I am going to call foo')
    foo()

print('I am outside both foo and bar')
bar()
print('I am outside both foo and bar')
```

- Functions can be defined inside other functions:

```python
def foo():
    def bar():
        print('bar is inside foo')
    bar()

foo()
```

Try calling `bar()` outside `foo`. What do you observe?

## Docstrings

Consider the following function:

```python
def square(x):
    """Return the square of x."""
    return x ** 2
```

The string immediately below the function definition is called a docstring. From the Python docs:

> A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.

Ignore unfamiliar terms such as "module" and "class". For now, it is sufficient to focus on functions. Adding the docstring to functions is a good practice. It may not be needed for simple and obvious functions like the one defined above. As the complexity of the functions you write increases, docstrings can be a life safer for other programmers reading your code.

The docstring associated with a given function can be accessed using the `__doc__` attribute:

```python
print(square.__doc__)
```

This gives `'Return the square of x.'` as output.