Python Functions

In this article, you'll learn about functions; what is a function, the syntax, components and types of a function. Also, you'll learn to create a function in Python.

In Python, function is a group of related statements that perform a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes code reusable.

Syntax of Function

```
def function_name(parameters):

        """docstring"""

        statement(s)
```

Above shown is a function definition which consists of following components.

1. Keyword `def` marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional `return` statement to return a value from the function.

Example of a function

```
def greet(name):
    """This function greets to
    the person passed in as
    parameter"""
    print("Hello, " + name + ". Good morning!")
```

Function Call

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')

Hello, Paul. Good morning!
```

Note: Try running the above code into the Python shell to see the output.

Docstring

The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does.

Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as `__doc__` attribute of the function.

For example:

Try running the following into the Python shell to see the output.

```
>>> print(greet.__doc__)
This function greets to
       the person passed into the
       name parameter
```

The return statement

The `return` statement is used to exit a function and go back to the place from where it was called.

Syntax of return

```
return [expression_list]
```

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the `return` statement itself is not present inside a function, then the function will return the `None` object.

For example:

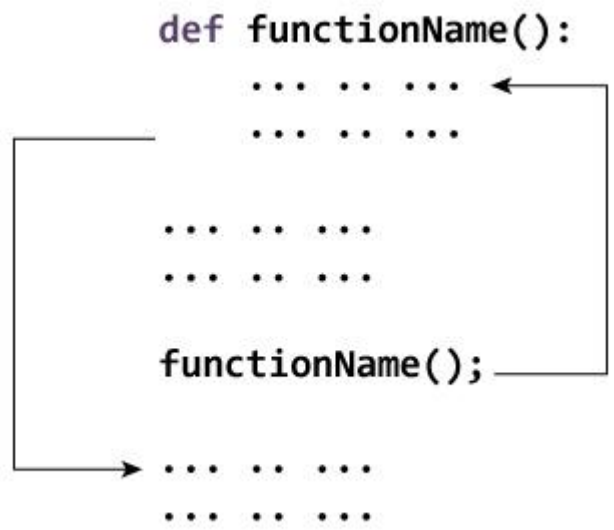
```
>>> print(greet("May"))
Hello, May. Good morning!
None
```

Here, `None` is the returned value.

Example of return

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""
    if num >= 0:
        return num
    else:
        return -num
# Output: 2
print(absolute_value(2))
# Output: 4
print(absolute_value(-4))
```

How Function works in Python?



Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

```
def my_func():
    x = 10
    print("Value inside function:",x)
x = 20
my_func()
print("Value outside function:",x)
```

Output

```
Value inside function: 10

Value outside function: 20
```

Here, we can see that the value of `x` is 20 initially. Even though the function `my_func()`changed the value of `x` to 10, it did not effect the value outside the function.

This is because the variable `x` inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword `global`.

Python Function Arguments

In Python, you can define a function that takes variable number of arguments. You will learn to define such functions using default, keyword and arbitrary arguments in this article.

In user-defined function topic, we learned about defining a function and calling it. Otherwise, the function call will result into an error. Here is an example.

```
def greet(name,msg):
   """This function greets to
   the person with the provided message"""
   print("Hello",name + ', ' + msg)
greet("Monica","Good morning!")
```

Output

```
Hello Monica, Good morning!
```

Here, the function `greet()` has two parameters.

Since, we have called this function with two arguments, it runs smoothly and we do not get any error.

If we call it with different number of arguments, the interpreter will complain. Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> greet("Monica")    # only one argument

TypeError: greet() missing 1 required positional argument: 'msg'

>>> greet()    # no arguments

TypeError: greet() missing 2 required positional arguments: 'name' and
'msg'
```

Variable Function Arguments

Up until now functions had fixed number of arguments. In Python there are other ways to define a function which can take variable number of arguments.

Three different forms of this type are described below.

Python Default Arguments

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=). Here is an example.

```
def greet(name, msg = "Good morning!"):
   """
   This function greets to
   the person with the
   provided message.
   If message is not provided,
   it defaults to "Good
   morning!"
   """
   print("Hello",name + ', ' + msg)
greet("Kate")
greet("Bruce","How do you do?")
```

In this function, the parameter `name` does not have a default value and is required (mandatory) during a call.

On the other hand, the parameter `msg` has a default value of `"Good morning!"`. So, it is optional during a call. If a value is provided, it will overwrite the default value.

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```

Python Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position.

For example, in the above function `greet()`, when we called it as `greet("Bruce","How do you do?")`, the value `"Bruce"` gets assigned to the argument `name` and similarly `"How do you do?"` to `msg`.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed. Following calls to the above function are all valid and produce the same result.

```
>>> # 2 keyword arguments

>>> greet(name = "Bruce",msg = "How do you do?")



>>> # 2 keyword arguments (out of order)

>>> greet(msg = "How do you do?",name = "Bruce")
```

```
>>> # 1 positional, 1 keyword argument

>>> greet("Bruce",msg = "How do you do?")
```

As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.

Having a positional argument after keyword arguments will result into errors. For example the function call as follows:

```
greet(name="Bruce","How do you do?")
```

Will result into error as:

```
SyntaxError: non-keyword arg after keyword arg
```

Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function.Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.

```
def greet(*names):
   """This function greets all
   the person in the names tuple."""
   # names is a tuple with arguments
   for name in names:
       print("Hello",name)
greet("Monica","Luke","Steve","John")
```

Output

```
Hello Monica
```

```
Hello Luke

Hello Steve

Hello John
```

Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a `for` loop to retrieve all the arguments back.

Types of Functions

Basically, we can divide functions into the following two types:

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

Python Recursion

In this article, you will learn to create a recursive function; a function that calls itself.

Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Python Recursive Function

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Following is an example of recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

Example of recursive function

```
# An example of a recursive function to
```

```
# find the factorial of a number
def calc_factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))
num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

In the above example, `calc_factorial()` is a recursive functions as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

```
calc_factorial(4)              # 1st call with 4
4 * calc_factorial(3)          # 2nd call with 3
4 * 3 * calc_factorial(2)      # 3rd call with 2
4 * 3 * 2 * calc_factorial(1)  # 4th call with 1
4 * 3 * 2 * 1                  # return from 4th call as number=1
4 * 3 * 2                      # return from 3rd call
4 * 6                          # return from 2nd call
24                             # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Advantages of recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Python Anonymous/Lambda Function

In this article, you'll learn about the anonymous function, also known as lambda functions. You'll learn what is it, its syntax and how to use it (with examples).

In Python, anonymous function is a function that is defined without a name.

While normal functions are defined using the `def` keyword, in Python anonymous functions are defined using the `lambda` keyword.

Hence, anonymous functions are also called lambda functions.

Lambda Functions

A lambda function has the following syntax.

Syntax of Lambda Function

```
lambda arguments: expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Example of Lambda Function

Here is an example of lambda function that doubles the input value.

```
# Program to show the use of lambda functions

double = lambda x: x * 2

# Output: 10

print(double(5))
```

In the above program, `lambda x: x * 2` is the lambda function. Here `x` is the argument and `x * 2` is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier `double`. We can now call it as a normal function. The statement

```
double = lambda x: x * 2
```

is nearly the same as

```
def double(x):

   return x * 2
```

Use of Lambda Function

We use lambda functions when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like `filter()`, `map()` etc.

Example use with filter()

The `filter()` function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluats to `True`.

Here is an example use of `filter()` function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

# Output: [4, 6, 8, 12]

print(new_list)
```

Example use with map()

The `map()` function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of `map()` function to double all the items in a list.

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

# Output: [2, 10, 8, 12, 16, 22, 6, 24]

print(new_list)
```

Python Modules

In this article, you will learn to create and import custom modules in Python. Also, you will find different techniques to import and use custom and built-in modules in Python.

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for e.g.: `example.py`, is called a module and its module name would be `example`.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as `example.py`.

```
# Python Module example


def add(a, b):
   """This program adds two
   numbers and return the result"""

   result = a + b
   return result
```

Here, we have defined a function `add()` inside a module named `example`. The function takes in two numbers and returns their sum.

How to import modules in Python?

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the `import` keyword to do this. To import our previously defined module `example` we type the following in the Python prompt.

```
>>> import example
```

This does not enter the names of the functions defined in `example` directly in the current symbol table. It only enters the module name `example` there.

Using the module name we can access the function using dot (.) operation. For example:

```
>>> example.add(4,5.5)

9.5
```

Python has a ton of standard modules available.

You can check out the full list of Python standard modules and what they are for. These files are in the Lib directory inside the location where you installed Python.

Standard modules can be imported the same way as we import our user-defined modules.

There are various ways to import modules. They are listed as follows.

Python import statement

We can import a module using `import` statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
# import statement example
# to import standard module math
import math
print("The value of pi is", math.pi)
```

When you run the program, the output will be:

```
The value of pi is 3.141592653589793
```

Import with renaming

We can import a module by renaming it as follows.

```
# import module by renaming it
import math as m
print("The value of pi is", m.pi)
```

We have renamed the `math` module as `m`. This can save us typing time in some cases.

Note that the name `math` is not recognized in our scope. Hence, `math.pi` is invalid, `m.pi` is the correct implementation.

Python from...import statement

We can import specific names from a module without importing the module as a whole. Here is an example.

```
# import only pi from math module
from math import pi
print("The value of pi is", pi)
```

We imported only the attribute pi from the module.

In such case we don't use the dot operator. We could have imported multiple attributes as follows.

```
>>> from math import pi, e

>>> pi

3.141592653589793

>>> e

2.718281828459045
```

Import all names

We can import all names(definitions) from a module using the following construct.

```
# import all names from the standard module math
from math import *
print("The value of pi is", pi)
```

We imported all the definitions from the math module. This makes all names except those beginnig with an underscore, visible in our scope.

Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module then (if not found) into a list of directories defined in `sys.path`. The search is in this order.

- The current directory.
- `PYTHONPATH` (an environment variable with a list of directory).
- The installation-dependent default directory.

```
>>> import sys

>>> sys.path

['',

'C:\\Python33\\Lib\\idlelib',

'C:\\Windows\\system32\\python33.zip',

'C:\\Python33\\DLLs',

'C:\\Python33\\lib',

'C:\\Python33',

'C:\\Python33\\lib\\site-packages']
```

We can add modify this list to add our own path.

Reloading a module

The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works.

Suppose we have the following code in a module named `my_module`.

```
# This module shows the effect of
#  multiple imports and reload

print("This code got executed")
```

Now we see the effect of multiple imports.

```
>>> import my_module

This code got executed

>>> import my_module

>>> import my_module
```

We can see that our code got executed only once. This goes to say that our module was imported only once.

Now if our module changed during the course of the program, we would have to reload it.One way to do this is to restart the interpreter. But this does not help much.

Python provides a neat way of doing this. We can use the `reload()` function inside the `imp` module to reload a module. This is how its done.

```
>>> import imp

>>> import my_module

This code got executed

>>> import my_module

>>> imp.reload(my_module)

This code got executed

<module 'my_module' from '.\\my_module.py'>
```

The dir() built-in function

We can use the `dir()` function to find out names that are defined inside a module.

For example, we have defined a function `add()` in the module `example` that we had in the beginning.

```
>>> dir(example)

['__builtins__',

'__cached__',

'__doc__',

'__file__',

'__initializing__',

'__loader__',

'__name__',

'__package__',

'add']
```

Here, we can see a sorted list of names (along with `add`). All other names that begin with an underscore are default Python attributes associated with the module (we did not define them ourself).

For example, the `__name__` attribute contains the name of the module.

```
>>> import example

>>> example.__name__

'example'
```

All the names defined in our current namespace can be found out using the `dir()` function without any arguments.

```
>>> a = 1

>>> b = "hello"

>>> import math

>>> dir()

['__builtins__', '__doc__', '__name__', 'a', 'b', 'math', 'pyscripter']
```

Python Package

In this article, you'll learn to divide your code base into clean, efficient modules using Python packages. Also, you'll learn to import and use your own or third party packages in your Python program.

We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access.

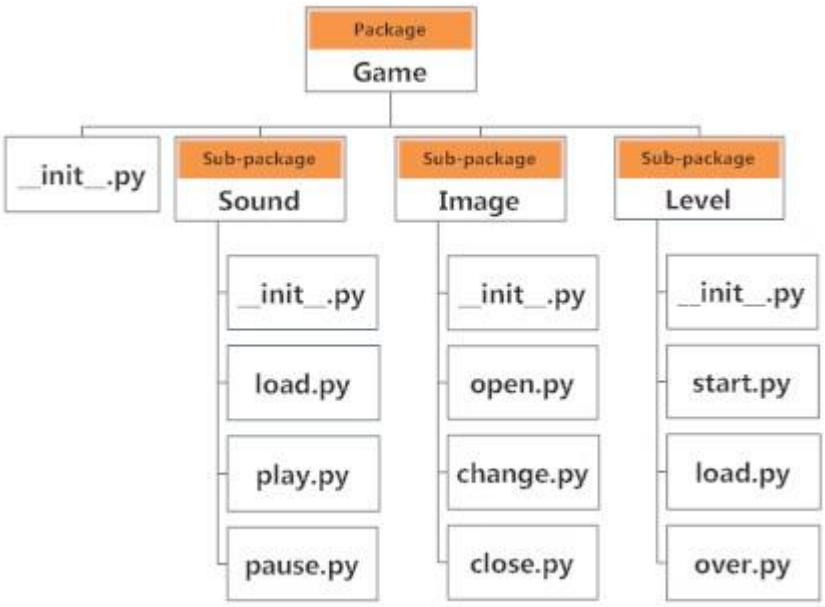
Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, **Python has packages for directories and modules for files.**

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.

A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.



Importing module from a package

We can import modules from packages using the dot (.) operator.

For example, if want to import the `start` module in the above example, it is done as follows.

```
import Game.Level.start
```

Now if this module contains a function named `select_difficulty()`, we must use the full name to reference it.

```
Game.Level.start.select_difficulty(2)
```

If this construct seems lengthy, we can import the module without the package prefix as follows.

```
from Game.Level import start
```

We can now call the function simply as foll ows.

```
start.select_difficulty(2)
```

Yet another way of importing just the required function (or class or variable) form a module within a package would be as follows.

```
from Game.Level.start import select_difficulty
```

Now we can directly call this function.

```
select_difficulty(2)
```

Although easier, this method is not recommended. Using the full namespace avoids confusion and prevents two same identifier names from colliding.

While importing packages, Python looks in the list of directories defined in `sys.path`, similar as for module search path.