Python Functions

programiz.com/python-programming/function

What is 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 samerules 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!")

How to call a function in python?

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!
```

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:

```
>>> 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

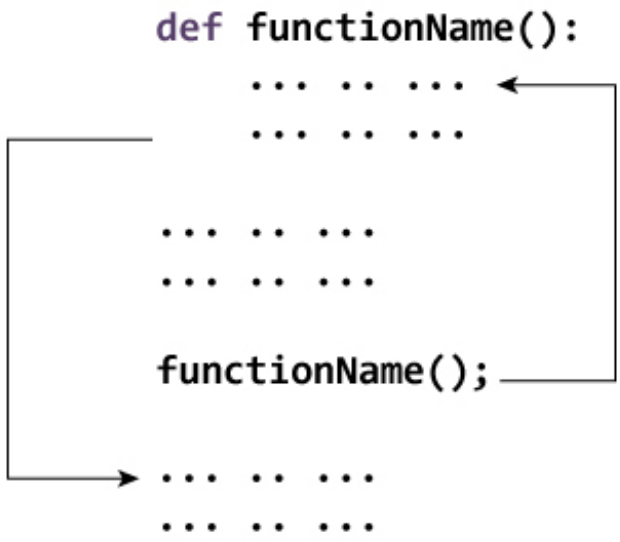
else:

return -num

print(absolute_value(2))

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`.

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 User-defined Functions

programiz.com/python-programming/user-defined-function

What are user-defined functions in Python?

Functions that we define ourselves to do certain specific task are referred as user-defined functions. The way in which we define and call functions in Python are already discussed.

Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of library, it can be termed as library functions.

All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmars working on large project can divide the workload by making different functions.

Example of a user-defined function

def add_numbers(x,y):

sum = x + y

return sum

num1 = 5

num2 = 6

print("The sum is", add_numbers(num1, num2))

Output

```
Enter a number: 2.4
Enter another number: 6.5
The sum is 8.9
```

Here, we have defined the function `my_addition()` which adds two numbers and returns the result.

This is our user-defined function. We could have multiplied the two numbers inside our function (it's all up to us). But this operation would not be consistent with the name of the function. It would create ambiguity.

It is always a good idea to name functions according to the task they perform.

In the above example, `input()`, `print()` and `float()` are built-in functions of the Python programming language.

Python Recursion

programiz.com/python-programming/recursion

What is recursion in Python?

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

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

programiz.com/python-programming/anonymous-function

What are lambda functions in Python?

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.

How to use lambda Functions in Python?

A lambda function in python has the following syntax.

Syntax of Lambda Function in python

```
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 in python

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

double = lambda x: x * 2

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 in python

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.

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

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

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.

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

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

print(new_list)
```

Python Global, Local and Nonlocal variables

programiz.com/python-programming/global-local-nonlocal-variables

Global Variables

In Python, a variable declared outside of the function or in global scope is known as global variable. This means, global variable can be accessed inside or outside of the function.

Let's see an example on how a global variable is created in Python.

Example 1: Create a Global Variable

```
x = "global"

def foo():

print("x inside :", x)

foo()

print("x outside:", x)
```

When we run the code, the will output be:

```
x inside : global
x outside: global
```

In above code, we created *x* as a global variable and defined a `foo()` to print the global variable *x*. Finally, we call the `foo()` which will print the value of *x*.

What if you want to change value of *x* inside a function?

```
x = "global"

def foo():

x = x * 2

print(x)

foo()
```

When we run the code, the will output be:

```
UnboundLocalError: local variable 'x' referenced before assignment
```

The output shows an error because Python treats *x* as a local variable and *x* is also not defined inside `foo()`.

To make this work we use `global` keyword, to learn more visit Python Global Keyword.

Local Variables

A variable declared inside the function's body or in the local scope is known as local variable.

Example 2: Accessing local variable outside the scope

```
def foo():

y = "local"

foo()

print(y)
```

When we run the code, the will output be:

```
NameError: name 'y' is not defined
```

The output shows an error, because we are trying to access a local variable *y* in a global scope whereas the local variable only works inside `foo()` or local scope.

Let's see an example on how a local variable is created in Python.

Example 3: Create a Local Variable

Normally, we declare a variable inside the function to create a local variable.

```
def foo():

y = "local"

print(y)

foo()
```

When we run the code, it will output:

```
local
```

Let's take a look to the earlier problem where *x* was a global variable and we wanted to modify *x* inside `foo()`.

Global and local variables

Here, we will show how to use global variables and local variables in the same code.

Example 4: Using Global and Local variables in same code

```
x = "global"

def foo():
```

```
global x

y = "local"

x = x * 2

print(x)

print(y)

foo()
```

When we run the code, the will output be:

```
global global
local
```

In the above code, we declare *x* as a global and *y* as a local variable in the `foo()`. Then, we use multiplication operator `*` to modify the global variable *x* and we print both *x* and *y*.

After calling the `foo()`, the value of *x* becomes `global global` because we used the `x * 2` to print two times `global`. After that, we print the value of local variable *y* i.e `local`.

Example 5: Global variable and Local variable with same name

```
x = 5

def foo():

x = 10

print("local x:", x)

foo()

print("global x:", x)
```

When we run the code, the will output be:

```
local x: 10
global x: 5
```

In above code, we used same name *x* for both global variable and local variable. We get different result when we print same variable because the variable is declared in both scopes, i.e. the local scope inside `foo()` and global scope outside `foo()`.

When we print the variable inside the `foo()` it outputs `local x: 10`, this is called local scope of variable.

Similarly, when we print the variable outside the `foo()`, it outputs `global x: 5`, this is called global scope of variable.

Nonlocal Variables

Nonlocal variable are used in nested function whose local scope is not defined. This means, the variable can be neither in the local nor the global scope.

Let's see an example on how a global variable is created in Python.

We use `nonlocal` keyword to create nonlocal variable.

Example 6: Create a nonlocal variable

12

def outer():

x = "local"

def inner():

nonlocal x

x = "nonlocal"

print("inner:", x)

inner()

print("outer:", x)

outer()

When we run the code, the will output be:

```
inner: nonlocal
outer: nonlocal
```

In the above code there is a nested function `inner()`. We use `nonlocal` keyword to create nonlocal variable. The `inner()` function is defined in the scope of another function `outer()`.

Note : If we change value of nonlocal variable, the changes appears in the local variable.

Python Global Keyword

programiz.com/python-programming/global-keyword

Introduction to global Keyword

In Python, `global` keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

Rules of global Keyword

The basic rules for `global` keyword in Python are:

- When we create a variable inside a function, it's local by default.
- When we define a variable outside of a function, it's global by default. You don't have to use `global` keyword.
- We use `global` keyword to read and write a global variable inside a function.
- Use of `global` keyword outside a function has no effect

Use of global Keyword (With Example)

Let's take an example.

Example 1: Accessing global Variable From Inside a Function

```
c = 1

def add():

print(c)

add()
```

When we run above program, the output will be:

```
1
```

However, we may have some scenarios where we need to modify the global variable from inside a function.

Example 2: Modifying Global Variable From Inside the Function

```
c = 1

def add():

c = c + 2
```

print(c)

add()

When we run above program, the output shows an error:

```
UnboundLocalError: local variable 'c' referenced before assignment
```

This is because we can only access the global variable but cannot modify it from inside the function.

The solution for this is to use the `global` keyword.

Example 3: Changing Global Variable From Inside a Function using global

c = 0

def add():

global c

c = c + 2

print("Inside add():", c)

add()

print("In main:", c)

When we run above program, the output will be:

```
Inside add(): 2
In main: 2
```

In the above program, we define *c* as a global keyword inside the `add()` function.

Then, we increment the variable *c* by `1`, i.e `c = c + 2`. After that, we call the `add()` function. Finally, we print global variable *c*.

As we can see, change also occured on the global variable outside the function, `c = 2`.

Global Variables Across Python Modules

In Python, we create a single module `config.py` to hold global variables and share information across Python modules within the same program.

Here is how we can share global variable across the python modules.

Example 4 : Share a global Variable Across Python Modules

Create a `config.py` file, to store global variables

```
a = 0
b = "empty"
```

Create a `update.py` file, to change global variables

```
import config

config.a = 10
config.b = "alphabet"
```

Create a `main.py` file, to test changes in value

```
import config
import update

print(config.a)
print(config.b)
```

When we run the `main.py` file, the output will be

```
10
alphabet
```

In the above, we create three files: `config.py`, `update.py` and `main.py`.

The module `config.py` stores global variables of *a* and *b*. In `update.py` file, we import the `config.py` module and modify the values of *a* and *b*. Similarly, in `main.py` file we import both `config.py` and `update.py` module. Finally, we print and test the values of global variables whether they are changed or not.

Global in Nested Functions

Here is how you can use a global variable in nested function.

Example 5: Using a Global Variable in Nested Function

def foo():

x = 20

def bar():

global x

x = 25

print("Before calling bar: ", x)

print("Calling bar now")

bar()

print("After calling bar: ", x)

foo()

print("x in main : ", x)

Powered by DataCamp

The output is :

```
Before calling bar: 20
Calling bar now
After calling bar: 20
x in main : 25
```

In the above program, we declare global variable inside the nested function `bar()`. Inside `foo()` function, *x* has no effect of global keyword.

Before and after calling `bar()`, the variable *x* takes the value of local variable i.e `x = 20`. Outside of the `foo()` function, the variable *x* will take value defined in the `bar()` function i.e `x = 25`. This is because we have used `global` keyword in *x* to create global variable inside the `bar()` function (local scope).

If we make any changes inside the `bar()` function, the changes appears outside the local scope, i.e. `foo()`.