**What is a function in Python?**

In Python, a function is a group of related statements that performs a specific task. a **function** is a self-contained block of code that encapsulates a specific task or related group of tasks

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 the code reusable.

### Abstraction and Reusability

Suppose you write some code that does something useful. As you continue development, you find that the task performed by that code is one you need often, in many different locations within your application. What should you do? Well, you could just replicate the code over and over again, using your editor’s copy-and-paste capability.

Later on, you’ll probably decide that the code in question needs to be modified. You’ll either find something wrong with it that needs to be fixed, or you’ll want to enhance it in some way. If copies of the code are scattered all over your application, then you’ll need to make the necessary changes in every location.

### Modularity

Functions allow **complex processes** to be broken up into smaller steps. Imagine, for example, that you have a program that reads in a file, processes the file contents, and then writes an output file. Your code could look like this:

def read\_file():

# Code to read file in

<statement>

<statement>

<statement>

<statement>

def process\_file():

# Code to process file

<statement>

<statement>

<statement>

<statement>

def write\_file():

# Code to write file out

<statement>

<statement>

<statement>

<statement>

# Main program

read\_file()

process\_file()

write\_file()

### Namespace Separation

A **namespace** is a region of a program in which **identifiers** have meaning. As you’ll see below, when a Python function is called, a new namespace is created for that function, one that is distinct from all other namespaces that already exist.

The practical upshot of this is that [variables](https://realpython.com/python-variables/) can be defined and used within a Python function even if they have the same name as variables defined in other functions or in the main program. In these cases, there will be no confusion or interference because they’re kept in separate namespaces.

This means that when you write code within a function, you can use variable names and identifiers without worrying about whether they’re already used elsewhere outside the function. This helps minimize errors in code considerably.

**Syntax of Function**

def function\_name(parameters):

"""docstring"""

statement(s)

Above shown is a function definition that consists of the following components.

1. Keyword def that marks the start of the function header.
2. A function name to uniquely identify the function. Function naming follows the same [rules of writing identifiers in Python](https://www.programiz.com/python-programming/keywords-identifier#rules).
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of the 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 the same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

**Example of a function**

def swap(x,y):

"""

This function greets to

the person passed in as

a parameter

"""

temp = x;

X=y;

Y=temp;

return(x)

return(y)

**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!

**Note:** Try running the above code in the Python program with the function definition to see the output.

def greet(name):

"""

This function greets to

the person passed in as

a parameter

"""

print("Hello, " + name + ". Good morning!")

greet('Paul')

**Docstrings**

The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain 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 the \_\_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 in as

a parameter

To learn more about docstrings in Python, visit [Python Docstrings](https://www.programiz.com/python-programming/docstrings).

**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 an expression that 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 since greet() directly prints the name and no return statement is used.

**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(“this is main program”)

x= absolute\_value(-5)

print(absolute\_value(2))

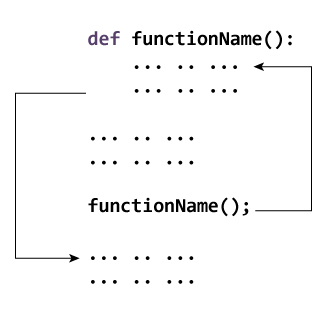
print(absolute\_value(-4))

**Output**

2

4

**How Function works in Python?**

Working of functions 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 are not visible from outside the function. Hence, they have a local scope.

The 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 affect 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 the same names, they are two different variables with different scopes.

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.

c = 0 # global variable

def add():

global c

c = c + 2 # increment by 2

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

add()

print("In main:", c)

**Types of Functions**

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

1. [Built-in functions](https://www.programiz.com/python-programming/methods/built-in) - Functions that are built into Python.
2. [User-defined functions](https://www.programiz.com/python-programming/user-defined-function) - Functions defined by the users themselves.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

* Function blocks begin with the keyword **def** followed by the function name and parentheses ( ( ) ).
* Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
* The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
* The code block within every function starts with a colon (:) and is indented.
* The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

def printme( str ):

"This prints a passed string into this function"

print str

return

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

[Live Demo](http://tpcg.io/3mrnYY)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme("I'm first call to user defined function!")

printme("Again second call to the same function")

When the above code is executed, it produces the following result −

I'm first call to user defined function!

Again second call to the same function

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

[Live Demo](http://tpcg.io/ZwGczd)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist.append([1,2,3,4]);

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

[Live Demo](http://tpcg.io/xgrZcs)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist = [1,2,3,4]; # This would assig new reference in mylist

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

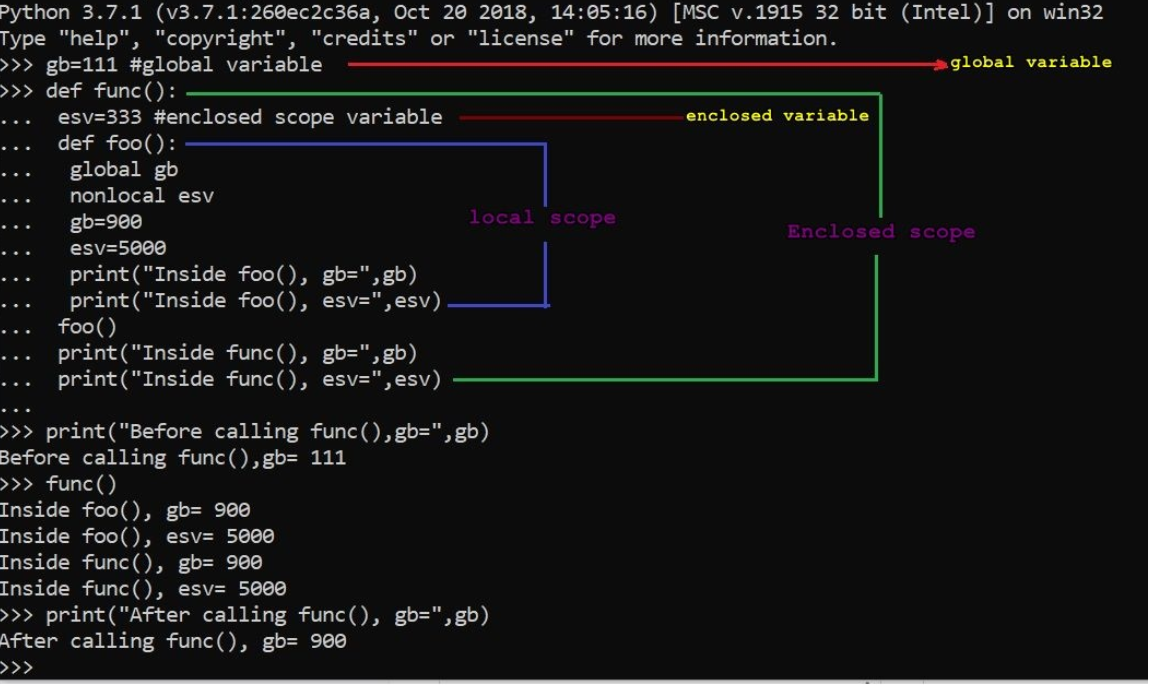
Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

Function Arguments

You can call a function by using the following types of formal arguments −

* Required arguments
* Keyword arguments
* Default arguments
* Variable-length arguments



Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

[Live Demo](http://tpcg.io/KIgYav)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme()

When the above code is executed, it produces the following result −

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

[Live Demo](http://tpcg.io/A0FJD1)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme( str = "My string")

When the above code is executed, it produces the following result −

My string

The following example gives more clear picture. Note that the order of parameters does not matter.

[Live Demo](http://tpcg.io/hdONo3)

#!/usr/bin/python

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age =50, name = "miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

[Live Demo](http://tpcg.io/JWyo6q)

#!/usr/bin/python

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Name: miki

Age 35

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

[Live Demo](http://tpcg.io/qofSxL)

#!/usr/bin/python

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

print "Output is: "

print arg1

for var in vartuple:

print var

return;

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

When the above code is executed, it produces the following result −

Output is:

10

Output is:

70

60

50

The *Anonymous* Functions

In Python, an anonymous function is a [function](https://www.programiz.com/python-programming/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.

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

* Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
* An anonymous function cannot be a direct call to print because lambda requires an expression
* Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
* Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how *lambda* form of function works −

[Live Demo](http://tpcg.io/TGEvCu)

#!/usr/bin/python

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )

When the above code is executed, it produces the following result −

Value of total: 30

Value of total: 40

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](https://www.programiz.com/python-programming/function-argument)).

### 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 evaluates 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))

print(new\_list)

**Output**

[4, 6, 8, 12]

### 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))

print(new\_list)

**Output**

[2, 10, 8, 12, 16, 22, 6, 24]

The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

[Live Demo](http://tpcg.io/FHnLKA)

#!/usr/bin/python

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print "Inside the function : ", total

return total;

# Now you can call sum function

total = sum( 10, 20 );

print "Outside the function : ", total

When the above code is executed, it produces the following result −

Inside the function : 30

Outside the function : 30

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

* Global variables
* Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

[Live Demo](http://tpcg.io/LFZtHG)

#!/usr/bin/python

total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2; # Here total is local variable.

print "Inside the function local total : ", total

return total;

# Now you can call sum function

sum( 10, 20 );

print "Outside the function global total : ", total

When the above code is executed, it produces the following result −

Inside the function local total : 30

Outside the function global total : 0

A recursive function is a function defined in terms of itself via self-referential expressions.

This means that the function will continue to call itself and repeat its behavior until some condition is met to [return](https://realpython.com/python-return-statement/) a result. All recursive functions share a common structure made up of two parts: base case and recursive case.

To demonstrate this structure, let’s write a recursive function for calculating n!:

1. Decompose the original problem into simpler instances of the same problem. This is the recursive case:
2. n! = n x (n−1) x (n−2) x (n−3) ⋅⋅⋅⋅ x 3 x 2 x 1
3. n! = n x (n−1)!
4. As the large problem is broken down into successively less complex ones, those subproblems must eventually become so simple that they can be solved without further subdivision. This is the base case:
5. n! = n x (n−1)!
6. n! = n x (n−1) x (n−2)!
7. n! = n x (n−1) x (n−2) x (n−3)!
8. ⋅
9. ⋅
10. n! = n x (n−1) x (n−2) x (n−3) ⋅⋅⋅⋅ x 3!
11. n! = n x (n−1) x (n−2) x (n−3) ⋅⋅⋅⋅ x 3 x 2!
12. n! = n x (n−1) x (n−2) x (n−3) ⋅⋅⋅⋅ x 3 x 2 x 1!

Here, 1! is our base case, and it equals 1.

Recursive function for calculating n! implemented in Python:

def factorial\_recursive(n):

# Base case: 1! = 1

if n == 1:

return 1

# Recursive case: n! = n \* (n-1)!

else:

return n \* factorial\_recursive(n-1)

>>>

>>> factorial\_recursive(5)

120

Care should be taken 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).

## Naive Recursion is Naive

The Fibonacci numbers were originally deﬁned by the Italian mathematician Fibonacci in the thirteenth century to model the growth of rabbit populations. Fibonacci surmised that the number of pairs of rabbits born in a given year is equal to the number of pairs of rabbits born in each of the two previous years, starting from one pair of rabbits in the ﬁrst year.

To count the number of rabbits born in the nth year, he deﬁned the recurrence relation:

Fn = Fn-1 + Fn-2

The base cases are:

F0 = 0 and F1 = 1

Let’s write a recursive function to compute the nth Fibonacci number:

def fibonacci\_recursive(n):

print("Calculating F", "(", n, ")", sep="", end=", ")

# Base case

if n == 0:

return 0

elif n == 1:

return 1

# Recursive case

else:

return fibonacci\_recursive(n-1) + fibonacci\_recursive(n-2)

>>>

>>> fibonacci\_recursive(5)

Calculating F(5), Calculating F(4), Calculating F(3), Calculating F(2), Calculating F(1),

Calculating F(0), Calculating F(1), Calculating F(2), Calculating F(1), Calculating F(0),

Calculating F(3), Calculating F(2), Calculating F(1), Calculating F(0), Calculating F(1),

5

Naively following the recursive deﬁnition of the nth Fibonacci number was rather inefficient. As you can see from the output above, we are unnecessarily recomputing values. Let’s try to improve fibonacci\_recursive by caching the results of each Fibonacci computation Fk:

from functools import lru\_cache

@lru\_cache(maxsize=None)

def fibonacci\_recursive(n):

print("Calculating F", "(", n, ")", sep="", end=", ")

# Base case

if n == 0:

return 0

elif n == 1:

return 1

# Recursive case

else:

return fibonacci\_recursive(n-1) + fibonacci\_recursive(n-2)

>>>

>>> fibonacci\_recursive(5)

Calculating F(5), Calculating F(4), Calculating F(3), Calculating F(2), Calculating F(1), Calculating F(0),

5

[lru\_cache](https://realpython.com/lru-cache-python/) is a [decorator](https://realpython.com/primer-on-python-decorators/) that caches the results. Thus, we avoid recomputation by explicitly checking for the value before trying to compute it. One thing to keep in mind about lru\_cache is that since it uses a dictionary to cache results, the positional and keyword arguments (which serve as keys in that dictionary) to the function must be hashable.