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](http://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 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!

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))

Output

2

4

# Program to illustrate

# the use of user-defined functions

def my\_addition(x,y):

"""This function adds two

numbers and return the result"""

sum = x + y

return sum

num1 = float(input("Enter a number: "))

num2 = float(input("Enter another number: "))

print("The sum is", my\_addition(num1,num2))

Output

Enter a number: 2.4

Enter another number: 6.5

The sum is 8.9

Explanation

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.

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.

## 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

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. But 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.

### 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)

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. Here are some valid calls to this function.

>>> greet("Kate")

Hello Kate, Good morning!

>>> greet("Bruce","How do you do?")

Hello Bruce, How do you do?

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

### 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 argumentname 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.

greet(name = "Bruce",msg = "How do you do?") # 2 keyword arguments

greet(msg = "How do you do?",name = "Bruce") # 2 keyword arguments (out of order)

greet("Bruce",msg = "How do you do?") # 1 positional, 1 keyword argument

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

### 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.

Python Recursive Function

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.

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 recur\_fact(x):

"""This is a recursive function

to find the factorial of an integer"""

if x == 1:

return 1

else:

return (x \* recur\_fact(x-1))

num = int(input("Enter a number: "))

if num >= 1:

print("The factorial of", num, "is", recur\_fact(num))

Output

Enter a number: 4

The factorial of 4 is 24

Explanation

In the above example, recur\_fact() 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.

recur\_fact(4) # 1st call with 4

4 \* recur\_fact(3) # 2nd call with 3

4 \* 3 \* recur\_fact(2) # 3rd call with 2

4 \* 3 \* 2 \* recur\_fact(1) # 4th call with 1

4 \* 3 \* 2 \* 1 # retrun 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. We must avoid infinite recursion.

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 Python, anonymous function is a [function](http://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.

## 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

print(double(5))

**Output**

10

**Explanation**

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 using filter() and

# lambda functions

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() and

# lambda functions

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]