III. Functions

1. What is a function?

Function is a group of related statements that perform a specific task.

Functions help in breaking 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""" # optional
      statement(s) # at-least one statement. If not implemented use pass
      return #(optional)
```

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.

```
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.

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.

```
>>> 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.

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

Here, `None` is the returned value as we don't have any return statement.

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
```

When we call the function, it returned the absolute(+ve) value of it.

```
print(absolute_value(208))
# Output: 208

print(absolute_value(-18))
# Output: 18
```

2.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 = 100
      print("Value inside function:",x)

x = 200
my_func()
print("Value outside function:",x)
```

Output

```
Value inside function: 100
Value outside function: 200
```

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 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.

3.Python Function Arguments

- A function can have any number of arguments. When calling a function, we should provide all the necessary arguments/parameters to run the execution smooth. Otherwise execution will throw with error mentioning **number of arguments not matched with the expected arguments count**.

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 variable function arguments.

- Python Default Arguments
- Python Keyword Arguments
- Python Arbitrary Arguments

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.

4. Python Recursive Function

We know that in Python, a function can call other functions. It is even possible for the function to call itself. This type of functions 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.

```
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.

5. Anonymous/Lambda function

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.

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

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

```
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 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 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))

# 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.

```
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)
```

Try executing and check how these below functions will accept the arguments.

```
# 1. Variable & formal arguments together
def test_var_args(farg, *args):
    print "formal arg:", farg
    for arg in args:
        print "another arg:", arg
```

```
test_var_args(1, "two", 3)

# 2. Keyword arguments & formal arguments togther
def test_var_kwargs(farg, **kwargs):
    print "formal arg:", farg
    for key in kwargs:
        print "another keyword arg: %s: %s" % (key, kwargs[key])

test_var_kwargs(farg=1, myarg2="two", myarg3=3)

# 3. Tuple and fixed arguments together.
def test_var_args_call(arg1, arg2, arg3):
    print "arg1:", arg1
    print "arg2:", arg2
    print "arg3:", arg3

args = ("two", 3)
test_var_args_call(1, *args)

# 4. Keywords arguments/Dict pass to a function.
def test_var_args_call(arg1, arg2, arg3):
    print "arg1:", arg1
    print "arg2:", arg2
    print "arg3:", arg3

kwargs = {"arg3": 3, "arg2": "two"}

test_var_args_call(1, **kwargs)
```

Exercise – 3

Write Functions for below requirements.

1. Check if given substring is part of string.
2. Find LCM of two given numbers.
3. Check if given number is prime
4. Check if given string is palindrome
5. Find the unique elements of a given list.
6. Find all duplicate elements of a given list.
7. Try variable & keyword arguments with multiple combinations.
8. Try modifying variable declared outside function and check if it's possible? Also, explore the possibilities if you want to modify.