Unit III: Functions, Objects and Graphics **Hours:15**

Functions: Program Routines- Defining Functions- More on Functions: Calling Value-Returning Functions- Calling Non-Value-Returning Functions- Parameter Passing - Keyword Arguments in Python - Default Arguments in Python-Variable Scope- Software Objects: What is an Object? Object References- Turtle Graphics – Turtle attributes.

Python Functions

It is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs,

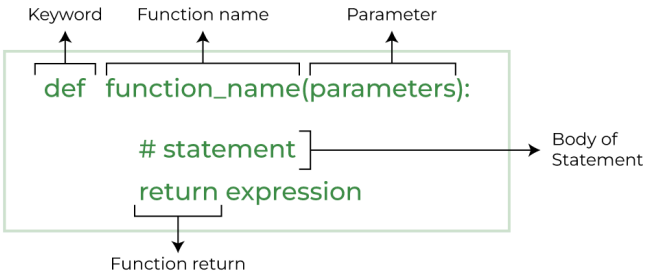
we can do the function calls to reuse code contained in it over and over again.

Some **Benefits of Using Functions**

- Increase Code Readability
- Increase Code Reusability

Python Function Declaration

The syntax to declare a function is:



- **Built-in library function:** These are Standard functions in Python that are available to use.
- **User-defined function:** We can create our own functions based on our requirements.

Creating a Function in Python

We can create a user-defined function in Python, using the **def** keyword. We can add any type of functionalities and properties to it as we require.

Python3

```
# A simple Python function

def fun():

 print("Welcome to GFG")
```

Calling a Python Function

After creating a function in Python we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.

Python3

```
# A simple Python function

def fun():

    print("Welcome to GFG")

 # Driver code to call a function

fun()
```

Output:
Welcome to GFG

Python Function with Parameters

If you have experience in C/C++ or Java then you must be thinking about the *return type* of the function and *data type* of arguments.

Defining and calling a function with parameters

```
def function_name(parameter: data_type) -> return_type:
    """Docstring"""
    # body of the function
    return expression
```

```
def add(num1: int, num2: int) -> int:

    """Add two numbers"""

    num3 = num1 + num2
```

```
    return num3

# Driver code

num1, num2 = 5, 15

ans = add(num1, num2)

print(f"The addition of {num1} and {num2} results {ans}.")
```

Output:
The addition of 5 and 15 results 20.

Return Statement in Python Function

The function return statement is used to exit from a function and go back to the function caller and return the specified value or data item to the caller.

The syntax for the return statement is:

return [expression_list]

The return statement can consist of a variable, an expression, or a constant which is returned at the end of the function execution. If none of the above is present with the return statement a None object is returned.

Example: Python Function Return Statement

```
def square_value(num):

    """This function returns the square

    value of the entered number"""

    return num**2

print(square_value(2))

print(square_value(-4))
```

Output:
4

16

Pass by Reference and Pass by Value

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created. Parameter passing in Python is the same as reference passing in Java.

```
# Here x is a new reference to same list lst

def myFun(x):

    x[0] = 20

 # Driver Code (Note that lst is modified

# after function call.

lst = [10, 11, 12, 13, 14, 15]

myFun(lst)

print(lst)
```

Output:
[20, 11, 12, 13, 14, 15]

Types of Python Function Arguments

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following 4 types of function arguments.

- **Default argument**
- **Keyword arguments (named arguments)**
- **Positional arguments**
- **Arbitrary arguments** (variable-length arguments *args and **kwargs)

Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments.

Python3

```
# Python program to demonstrate

# default arguments
```

```
def myFun(x, y=50):

    print("x: ", x)

    print("y: ", y)

  # Driver code (We call myFun() with only

# argument)

myFun(10)
```

Output:
x: 10

y: 50

Keyword Arguments

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

Python3

```
# Python program to demonstrate Keyword Arguments

def student(firstname, lastname):

    print(firstname, lastname)

 # Keyword arguments

student(firstname='Geeks', lastname='Practice')

student(lastname='Practice', firstname='Geeks')
```

Output:
Geeks Practice

Geeks Practice

Positional Arguments

We used the Position argument during the function call so that the first argument (or value) is assigned to name and the second argument (or value) is assigned to age.

By changing the position, or if you forget the order of the positions, the values can be used in the wrong places, as shown in the Case-2 example below, where 27 is assigned to the name and Suraj is assigned to the age.

Python3

```
def nameAge(name, age):

    print("Hi, I am", name)

    print("My age is ", age)

 # You will get correct output because

# argument is given in order

print("Case-1:")

nameAge("Suraj", 27)

# You will get incorrect output because

# argument is not in order

print("\nCase-2:")

nameAge(27, "Suraj")
```

Output:

Case-1:

Hi, I am Suraj
My age is 27

Case-2:

Hi, I am 27
My age is Suraj

Arbitrary Keyword Arguments

In Python Arbitrary Keyword Arguments, *args, and **kwargs can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- *args in Python (Non-Keyword Arguments)
- **kwargs in Python (Keyword Arguments)

Example 1: Variable length non-keywords argument

Python3

```
# Python program to illustrate

# *args for variable number of arguments

def myFun(*argv):

    for arg in argv:

        print(arg)

 myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output:

Hello

Welcome

to

GeeksforGeeks

Python Variable Scope

In Python, we can declare variables in three different scopes: local scope, global, and nonlocal scope.

A variable scope specifies the region where we can access a variable. For example,

```
def add_numbers():
    sum = 5 + 4
```

Based on the scope, we can classify Python variables into three types:

1. Local Variables
2. Global Variables
3. Nonlocal Variables

Python Local Variables

When we declare variables inside a function, these variables will have a local scope (within the function). We cannot access them outside the function.

These types of variables are called local variables. For example,

```
def greet():

    # local variable
    message = 'Hello'

    print('Local', message)

greet()

# try to access message variable
# outside greet() function
print(message)
```

Output

```
Local Hello
NameError: name 'message' is not defined
```

Python Global Variables

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

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

```
# declare global variable
message = 'Hello'

def greet():
    # declare local variable
    print('Local', message)

greet()
print('Global', message)
```

Output

```
Local Hello
Global Hello
```

Python Nonlocal Variables

In Python, nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.

We use the nonlocal keyword to create nonlocal variables.For example,

```
# outside function
```

```
def outer():
    message = 'local'

    # nested function
    def inner():

        # declare nonlocal variable
        nonlocal message

        message = 'nonlocal'
        print("inner:", message)

    inner()
    print("outer:", message)

outer()
```

Output

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

object in Python

Objects are variables that contain data and functions that can be used to manipulate the data.

The object's data can vary in type (string, integer, etc.) depending on how it's been defined.

An object is like a mini-program inside python, with its own set of rules and behaviors.

Creating an Object of class

The object is essential to work with the class attributes.

Instantiate is a term used when we create the object of any class, and the instance is also referred to as an object.

The object is created using the class name. The syntax is given below.

Syntax:

1. <object-name> = <**class**-name>(<arguments>)

```
2. class Person:
3.     name = "John"
4.     age = 24
5.     def display (self):
6.         print("Age: %d \nName: %s"%(self.age,self.name))
7. # Creating a emp instance of Employee class
8. per = Person()
9. per.display()
```

Output:

```
Age: 24
Name: John
```

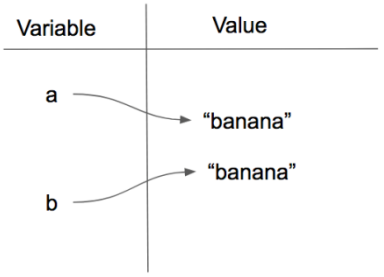
Objects and References

If we execute these assignment statements,

```
a = "banana"
b = "banana"
```

we know that a and b will refer to a string with the letters "banana". But we don't know yet whether they point to the *same* string.

There are two possible ways the Python interpreter could arrange its internal states:



In one case, a and b refer to two different string objects that have the same value. In the second case, they refer to the same object. Remember that an object is something a variable can refer to.

Ex: a = "banana"

b = "banana"

print(a is b)

output: True

Ex: a = [81,82,83]

b = [81,82,83]

print(a is b)

print(a == b)

print(id(a))

print(id(b))

output:

True

False

1

2

Turtle Programming in Python

Turtle" is a Python feature like a drawing board, which lets us command a turtle to draw all over it! We can use functions like turtle.forward(…) and turtle.right(…) which can move the turtle around. Commonly used turtle methods are :

Method	Parameter	Description
Turtle()	None	Creates and returns a new turtle object
forward()	amount	Moves the turtle forward by the specified amount
backward()	amount	Moves the turtle backward by the specified amount
right()	angle	Turns the turtle clockwise
left()	angle	Turns the turtle counterclockwise
penup()	None	Picks up the turtle's Pen
pendown()	None	Puts down the turtle's Pen
up()	None	Picks up the turtle's Pen
down()	None	Puts down the turtle's Pen
color()	Color name	Changes the color of the turtle's pen
fillcolor()	Color name	Changes the color of the turtle will use to fill a polygon
heading()	None	Returns the current heading
position()	None	Returns the current position
goto()	x, y	Move the turtle to position x,y
begin_fill()	None	Remember the starting point for a filled polygon

Method	Parameter	Description
end_fill()	None	Close the polygon and fill with the current fill color
dot()	None	Leave the dot at the current position
stamp()	None	Leaves an impression of a turtle shape at the current location
shape()	shapename	Should be ‘arrow’, ‘classic’, ‘turtle’ or ‘circle’

Plotting using Turtle

To make use of the turtle methods and functionalities, we need to import turtle.

”turtle” comes packed with the standard Python package and need not be installed externally.

The roadmap for executing a turtle program follows 4 steps:

1. Import the turtle module
2. Create a turtle to control.
3. Draw around using the turtle methods.
4. Run turtle.done().

```
from turtle import *
# or
import turtle
```

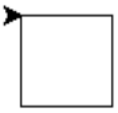
Shape 1: Square

Python

```
# Python program to draw square

# using Turtle Programming

import turtle

skk = turtle.Turtle()

 for i in range(4):

    skk.forward(50)
```

```
    skk.right(90)

     turtle.done()
```

Output:



Python3

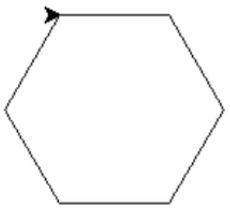
```
# Python program to draw star

# using Turtle Programming

import turtle

star = turtle.Turtle()

 star.right(75)

star.forward(100)

 for i in range(4):

    star.right(144)

    star.forward(100)

    turtle.done()
```

Output:

Shape 3: Hexagon
Python

```
# Python program to draw hexagon

# using Turtle Programming

import turtle

polygon = turtle.Turtle()

 num_sides = 6

side_length = 70

angle = 360.0 / num_sides

 for i in range(num_sides):

    polygon.forward(side_length)

    polygon.right(angle)

    turtle.done()
```

Output:



Changing the Screen Color

By default, the turtle screen is opened with the white background. However, we can modify the background color of the screen using the following function.

Example -

```
1. import turtle
2. # Creating turtle screen
```

3. t = turtle.Turtle()
4.
5. turtle.bgcolor("red")
6.
7. turtle.mainloop()

Output:

Turtle Attributes:

- **position()** -provides a tuple of coordinates as the current position of the turtle.
- **heading()** -returns the current heading angle of the turtle in degrees.
- **color()** -returns a tuple of RGB values representing the turtle's current color.
- **pensize()** -returns the current pen size of the turtle.
- **speed()** -returns the current speed of the turtle.