Unit 3 : Python Functions, Modules and Packages

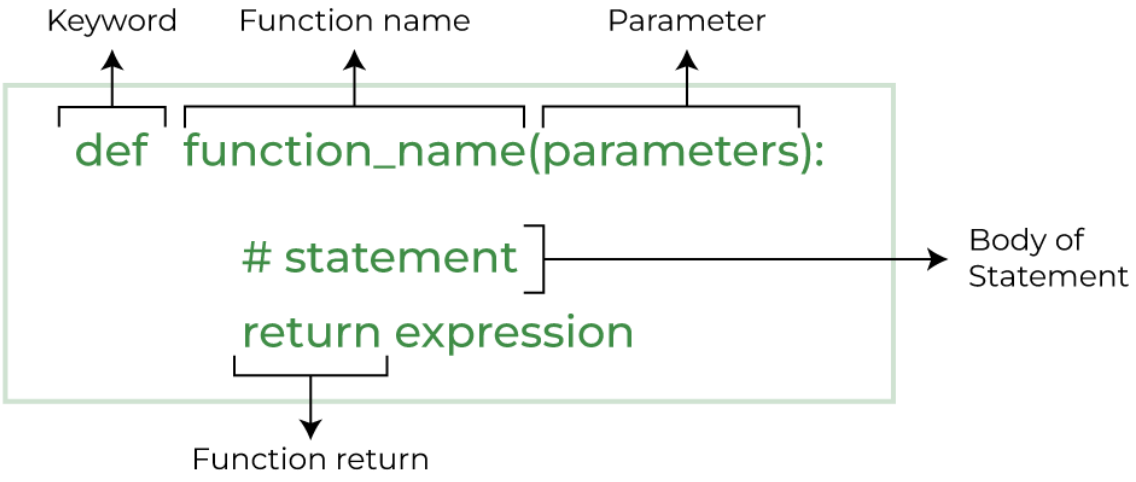
Unit 3: Python Functions, Modules and Packages

Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Python Function Declaration

The syntax to declare a function is:



Syntax of Python Function Declaration

Types of Functions in Python

Below are the different types of functions in **Python**:

- **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.

Types of Functions in Python

- **Default Argument**
  - 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 to write functions in Python.

```
# 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 (named 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.

```
# Python program to demonstrate Keyword Arguments
def student(firstname, lastname):
    print(firstname, lastname)
```

```
# Keyword arguments
student(firstname='Milan', lastname='Poria')
student(lastname='Poria', firstname='Milan')
```

Output:

```
Milan Poria
Milan Poria
```

- 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.

```
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)


# You will get correct output because# argument is given
in orderprint("Case-1:")
nameAge("Suraj", 27)
# You will get incorrect output because# argument is not
in orderprint("\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 Arguments (variable-length arguments *args and **kwargs)**
  - 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

```
# 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
```

Example 2: Variable length keyword arguments:

```
# Python program to illustrate
# *kwargs for variable number of keyword arguments


def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))
```

```
# Driver code
myFun(first='Geeks', mid='for', last='Geeks')
```

Output:

```
first == Geeks
mid == for
last == Geeks
```

- **Anonymous Functions (lambda)**

  In Python, an **anonymous function** means that a function is without a name. As we already know the def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.

```
# Python code to illustrate the cube of a number
# using lambda function
def cube(x): return x*x*x

cube_v2 = lambda x : x*x*x

print(cube(7))
print(cube_v2(7))
```

Output:

```
343
343
```

- **Recursive Functions**
  - **Recursion** in Python refers to when a function calls itself. There are many instances when you have to build a recursive function to solve **Mathematical and Recursive Problems.** Using a recursive function should be done with caution, as a recursive function can

become like a non-terminating loop. It is better to check your exit statement while creating a recursive function.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(4))
```

Output

```
24
```

- 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 Python, 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]
```

When we pass a reference and change the received reference to something else, the connection between the passed and received parameters is broken.

For example, consider the below program as follows:

```
def myFun(x):

    # After below line link of x with previous
    # object gets broken. A new object is assigned
    # to x.
    x = [20, 30, 40]


# Driver Code (Note that lst is not modified
# after function call.
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
print(lst)
```

Output:

```
[10, 11, 12, 13, 14, 15]
```

Python Modules

- A Python module is a file containing Python definitions and statements.
- A module can define functions, classes, and variables.
- A module can also include runnable code.
- Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.
- To create a Python module, write the desired code and save that in a file with a .py extension.
- The functions and classes defined in a module can be imported to another module using the import statement in some other Python source file.
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

Locating Python Modules

- Whenever a module is imported in Python, the interpreter looks in several locations.
- First, it will check for the built-in module; if not found, then it looks in a list of directories defined in the sys.path.
- Python interpreter searches for the module in the following manner:
  1. First, it searches for the module in the current directory.
  2. If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH.
  3. If that also fails, Python checks the installation-dependent list of directories configured at the time Python is installed.

Example:

Create a Python Module

To create a Python module, write the desired code and save that in a file with **.py** extension. Let's understand it better with an example:

Example:

Let's create a simple calc.py in which we define two functions, one **add** and another **subtract**.

Python

Explain

```
# A simple module, calc.pydef add(x, y):
return (x+y)

def subtract(x, y):
return (x-y)
```

Import module in Python

We can import the functions, and classes defined in a module to another module using the **import statement** in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

***Note**: A search path is a list of directories that the interpreter searches for importing a module.*

For example, to import the module calc.py, we need to put the following command at the top of the script.

Syntax to Import Module in Python

```
import module
```

Note: This does not import the functions or classes directly instead imports the module only. To access the functions inside the module the dot(.) operator is used.

Importing modules in Python Example

Now, we are importing the **calc** that we created earlier to perform add operation.

Python

```
# importing  module calc.py import  calc print(calc.add(10, 2))
```

Output:

```
12
```

Packages in Python

- Python Packages are a way to organize and structure your Python code into reusable components.
- A package is like a folder that contains related Python files (modules) that work together to provide certain functionality.
- Packages help keep your code organized, make it easier to manage and maintain, and allow you to share your code with others.

Creating Packages in Python

1. **Create a Directory**: Start by creating a directory (folder) for your package. This directory will serve as the root of your package structure.

2. **Add Modules**: Within the package directory, add Python files (modules) containing your code. Each module should represent a distinct functionality or component of your package.
3. **Init File**: Include an `__init__.py` file in the package directory. This file can be empty or contain initialization code for your package. It signals to Python that the directory should be treated as a package.
4. **Subpackages**: You can create sub-packages within your package by adding additional directories containing modules, along with their own `__init__.py` files.

Importing

- To use modules from your package, import them into your Python scripts using dot notation.
    - Example: If you have a module named `module1.py` inside a package named `mypackage`, you would import its function like this: `from mypackage.module1 import greet`.

Distribution

- If you want to distribute your package for others to use, you can create a `setup.py` file using Python's setuptools library. This file defines metadata about your package and specifies how it should be installed.

Example:

Here's a basic code sample demonstrating how to create a simple Python package:

1. Create a directory named mypackage.
2. Inside mypackage, create two Python files: module1.py and module2.py.
3. Create an __init__.py file inside mypackage (it can be empty).
4. Add some code to the modules.
5. Finally, demonstrate how to import and use the modules from the package.

```
mypackage/
│
├── __init__.py
```

```
├── module1.py
└── module2.py
```

Example: Now, let's create a Python script outside the mypackage directory to import and use these modules:

Explain

```
# module1.py
def greet(name):
    print(f"Hello, {name}!")
```

```
# module2.py
def add(a, b):
    return a + b
```

```
from mypackage import module1, module2

# Using functions from module1
module1.greet("Alice")

# Using functions from module2
result = module2.add(3, 5)
print("The result of addition is:", result)
```

When you run the script, you should see the following output:

```
Hello, Alice!
The result of addition is: 8
```