Python - Modules

Python Modules

The concept of **module** in Python further enhances the modularity. You can define more than one related functions together and load required functions. A module is a file containing definition of functions, classes, variables, constants or any other Python object. Contents of this file can be made available to any other program. Python has the **import** keyword for this purpose.

> *A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.*

Example of Python Module

</> Open Compiler

```
import math
print ("Square root of 100:", math.sqrt(100))
```

It will produce the following **output** –

```
Square root of 100: 10.0
```

Python Built-in Modules

Python's standard library comes bundled with a large number of modules. They are called built-in modules. Most of these built-in modules are written in C (as the reference implementation of Python is in C), and pre-compiled into the library. These modules pack useful functionality like system-specific OS management, disk IO, networking, etc.

Here is a select list of built-in modules –

Sr.No.	Name & Brief Description

1	**os** This module provides a unified interface to a number of operating system functions.
2	**string** This module contains a number of functions for string processing
3	**re** This module provides a set of powerful regular expression facilities. Regular expression (RegEx), allows powerful string search and matching for a pattern in a string
4	**math** This module implements a number of mathematical operations for floating point numbers. These functions are generally thin wrappers around the platform C library functions.
5	**cmath** This module contains a number of mathematical operations for complex numbers.
6	**datetime** This module provides functions to deal with dates and the time within a day. It wraps the C runtime library.
7	**gc** This module provides an interface to the built-in garbage collector.
8	**asyncio** This module defines functionality required for asynchronous processing
9	**Collections** This module provides advanced Container datatypes.
10	**functools** This module has Higher-order functions and operations on callable objects. Useful in functional programming
11	**operator** Functions corresponding to the standard operators.
12	**pickle** Convert Python objects to streams of bytes and back.
13	**socket** Low-level networking interface.
14	**sqlite3**

	A DB-API 2.0 implementation using SQLite 3.x.
15	**statistics** Mathematical statistics functions
16	**typing** Support for type hints
17	**venv** Creation of virtual environments.
18	**json** Encode and decode the JSON format.
19	**wsgiref** WSGI Utilities and Reference Implementation.
20	**unittest** Unit testing framework for Python.
21	**random** Generate pseudo-random numbers
22	**sys** Provides functions that acts strongly with the interpreter.
23	**requests** It simplifies HTTP requests by offering a user-friendly interface for sending and handling responses.

Python User-defined Modules

Any text file with **.py** extension and containing Python code is basically a module. It can contain definitions of one or more functions, variables, constants as well as classes. Any Python object from a module can be made available to interpreter session or another Python script by import statement. A module can also include runnable code.

Creating a Python Module

Creating a module is nothing but saving a Python code with the help of any editor. Let us save the following code as **mymodule.py**

```
def SayHello(name):
   print ("Hi {}! How are you?".format(name))
```

```
   return
```

You can now import mymodule in the current Python terminal.

```
>>> import mymodule
>>> mymodule.SayHello("Harish")
Hi Harish! How are you?
```

You can also import one module in another Python script. Save the following code as example.py

```
import mymodule
mymodule.SayHello("Harish")
```

Run this script from command terminal

```
Hi Harish! How are you?
```

The import Statement

In Python, the **import** keyword has been provided to load a Python object from one module. The object may be a function, class, a variable etc. If a module contains multiple definitions, all of them will be loaded in the namespace.

Let us save the following code having three functions as **mymodule.py.**

```
def sum(x,y):
   return x+y

def average(x,y):
   return (x+y)/2

def power(x,y):
   return x**y
```

The **import mymodule** statement loads all the functions in this module in the current namespace. Each function in the imported module is an attribute of this module object.

```
>>> dir(mymodule)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
```

```
'__package__', '__spec__', 'average', 'power', 'sum']
```

To call any function, use the module object's reference. For example, mymodule.sum().

```
import mymodule
print ("sum:",mymodule.sum(10,20))
print ("average:",mymodule.average(10,20))
print ("power:",mymodule.power(10, 2))
```

It will produce the following **output** –

```
sum:30
average:15.0
power:100
```

The from ... import Statement

The import statement will load all the resources of the module in the current namespace. It is possible to import specific objects from a module by using this syntax. For example –

Out of three functions in **mymodule**, only two are imported in following executable script **example.py**

```
from mymodule import sum, average
print ("sum:",sum(10,20))
print ("average:",average(10,20))
```

It will produce the following output –

```
sum: 30
average: 15.0
```

Note that function need not be called by prefixing name of its module to it.

The from...import * Statement

It is also possible to import all the names from a module into the current namespace by using the following import statement –

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

The import ... as Statement

You can assign an alias name to the imported module.

```
from modulename as alias
```

The **alias** should be prefixed to the function while calling.

Take a look at the following **example** –

```
import mymodule as x
print ("sum:",x.sum(10,20))
print ("average:", x.average(10,20))
print ("power:", x.power(10, 2))
```

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences –

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The PYTHONPATH Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system –

```
set PYTHONPATH = c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system −

```
set PYTHONPATH = /usr/local/lib/python
```

Namespaces and Scoping

Variables are names (identifiers) that map to objects. A namespace is a dictionary of variable names (keys) and their corresponding objects (values).

- A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable.
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.
- In order to assign a value to a global variable within a function, you must first use the global statement.
- The statement global VarName tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

Example

For example, we define a variable Money in the global namespace. Within the function Money, we assign Money a value, therefore Python assumes Money as a local variable. However, we accessed the value of the local variable Money before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

</> Open Compiler

```
Money = 2000
def AddMoney():
   # Uncomment the following line to fix the code:
   # global Money
   Money = Money + 1

print (Money)
AddMoney()
print (Money)
```

Module Attributes

In Python, a module is an object of module class, and hence it is characterized by attributes.

Following are the module attributes –

- __file__ returns the physical name of the module.
- __package__ returns the package to which the module belongs.
- __doc__ returns the docstring at the top of the module if any
- __dict__ returns the entire scope of the module
- __name__ returns the name of the module

Example

Assuming that the following code is saved as **mymodule.py**

```
"The docstring of mymodule"
def sum(x,y):
   return x+y

def average(x,y):
   return (x+y)/2

def power(x,y):
   return x**y
```

Let us check the attributes of mymodule by importing it in the following script –

```
import mymodule

print ("__file__ attribute:", mymodule.__file__)
print ("__doc__ attribute:", mymodule.__doc__)
print ("__name__ attribute:", mymodule.__name__)
```

It will produce the following **output** –

```
__file__ attribute: C:\math\examples\mymodule.py
__doc__ attribute: The docstring of mymodule
```

```
__name__ attribute: mymodule
```

The __name__Attribute

The __name__ attribute of a Python module has great significance. Let us explore it in more detail.

In an interactive shell, __name__ attribute returns '__main__'

```
>>> __name__
'__main__'
```

If you import any module in the interpreter session, it returns the name of the module as the __name__ attribute of that module.

```
>>> import math
>>> math.__name__
'math'
```

From inside a Python script, the __name__ attribute returns '__main__'

```
#example.py
print ("__name__ attribute within a script:", __name__)
```

Run this in the command terminal –

```
__name__ attribute within a script: __main__
```

This attribute allows a Python script to be used as executable or as a module. Unlike in C++, Java, C# etc., in Python, there is no concept of the main() function. The Python program script with .py extension can contain function definitions as well as executable statements.

Save **mymodule.py** and with the following code –

</> Open Compiler

```
"The docstring of mymodule"
def sum(x,y):
   return x+y
```

```
print ("sum:",sum(10,20))
```

You can see that sum() function is called within the same script in which it is defined.

```
sum: 30
```

Now let us import this function in another script **example.py**.

```
import mymodule
print ("sum:",mymodule.sum(10,20))
```

It will produce the following **output** –

```
sum: 30
sum: 30
```

The output "sum:30" appears twice. Once when mymodule module is imported. The executable statements in imported module are also run. Second output is from the calling script, i.e., **example.py** program.

What we want to happen is that when a module is imported, only the function should be imported, its executable statements should not run. This can be done by checking the value of __name__. If it is __main__, means it is being run and not imported. Include the executable statements like function calls conditionally.

Add **if** statement in **mymodule.py** as shown –

</> Open Compiler

```
"The docstring of mymodule"
def sum(x,y):
   return x+y

if __name__ == "__main__":
   print ("sum:",sum(10,20))
```

Now if you run **example.py** program, you will find that the sum:30 output appears only once.

```
sum: 30
```

The dir( ) Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example −

</> Open Compiler

```
# Import built-in module math
import math

content = dir(math)
print (content)
```

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

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

The reload() Function

Sometimes you may need to reload a module, especially when working with the interactive interpreter session of Python.

Assume that we have a test module (test.py) with the following function −

```
def SayHello(name):
   print ("Hi {}! How are you?".format(name))
   return
```

We can import the module and call its function from Python prompt as −

```
>>> import test
>>> test.SayHello("Deepak")
Hi Deepak! How are you?
```

However, suppose you need to modify the SayHello() function, such as −

```
def SayHello(name, course):
   print ("Hi {}! How are you?".format(name))
   print ("Welcome to {} Tutorial by TutorialsPoint".format(course))
   return
```

Even if you edit the test.py file and save it, the function loaded in the memory won't update. You need to reload it, using reload() function in imp module.

```
>>> import imp
>>> imp.reload(test)
>>> test.SayHello("Deepak", "Python")
Hi Deepak! How are you?
Welcome to Python Tutorial by TutorialsPoint
```

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules, subpackages and, sub-subpackages, and so on.

Consider a file Pots.py available in Phone directory. This file has following line of source code −

```
def Pots():
   print "I'm Pots Phone"
```

Similar way, we have another two files having different functions with the same name as above −

- Phone/Isdn.py file having function Isdn()
- Phone/G3.py file having function G3()

Now, create one more file __init__.py in Phone directory −

- Phone/__init__.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as follows –

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

After you add these lines to __init__.py, you have all of these classes available when you import the Phone package.

```
# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

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

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.