
Python Programming
High performance. Delivered.
Module 09 - Modules & Packages
accenture
Strategy | Digital | Technology | Operations

Agenda

- **Modules**
- **Packages**

Module Objectives

At the end of this module, you will be able to understand

Modules

- Creating a Module
- Different ways on importing a module
- Module Search Path (Locating Modules)
- Reloading a module
- dir() built-in function
- *PYTHONPATH* Variable
- Namespaces and Scoping
- *globals()* and *locals()* Functions

Packages

- Creating a Package
- Importing a module from a package

Modules in Python

A Module in Python refers to a file containing

- statements
- definitions.

Example:

MyModule.py

Here “MyModule” is the name of the module.

Uses of Module

Modules are used to

- breakdown large programs into small manageable and organized files.
- facilitate reusability of code.

Modules can be defined and later imported into any python program.

Creating a Module in Python

Modules are created by creating a file with extension .py and storing functions , statements and variables.

Example:

Create a file “mymodule.py” and enter the following code…

```
def function1(a, b):
    """This program adds two numbers and return the result"""
    result = a + b
    return result

def function2(a, b):
    """This program subtracts two numbers and return the result"""
    result = a - b
    return result

MyVar = 345;     # Variable MyVar is defined and initialised
```

SAVE the file.

Note: mymodule is created.

Importing modules

Once the module is created , it should be imported inside another module or the interactive interpreter in Python.

To do that , import keyword is used.

Example:

import mymodule

To call the function defined inside the module, we need to use the dot (.) operator.

Example:

mymodule.function1(10,30);

Different ways to import modules

Module can be imported using the following ways

- The import statement
- Import with renaming statement
- from...import statement
- Import all names statement

The import statement

We can import a module using import statement and access the definitions inside it using the dot operator.

Use the module mymodule created earlier

```
import mymodule;

sum = mymodule.function1(10,30);
print("SUM = ",sum);

diff = mymodule.function2(40,30);
print("DIFFERENCE = ",diff);
```

OUTPUT

```
SUM =  40
DIFFERENCE =  10
```

Import with renaming statement

We can import a module by renaming it as follows.

Example:

```
import mymodule as mm;          # the module mymodule is renamed as mm

sum = mm.function1(10,30);
print("SUM = ",sum);

diff = mm.function2(40,30);
print("DIFFERENCE = ",diff);
```

Note: once the module is renamed, the old name is not recognized. Hence mymodule.function1(10,30) would be an invalid statement.

```
OUTPUT
SUM =  40
DIFFERENCE =  10
```

The from...import statement

We can import specific names form a module without importing the module as a whole.

Example:

```
from mymodule import MyVar;      # import only MyVar from mymodule

print("The value of MyVar = ", MyVar)
```

```
OUTPUT
The value of MyVar =  345
```

Import all names statement

We can import all names(definitions) form a module using the following construct.

Example:

```
from mymodule import *;      # import all the entities  from mymodule

sum = function1(10,30);
print("SUM = ",sum);

diff = function2(40,30);
print("DIFFERENCE = ",diff);

print("The value of MyVar = ", MyVar);
```

'*' makes all names except those beginning with an underscore, visible in our scope.

Note:

- Importing everything with the asterisk (*) symbol is not a good programming practice.
- It can lead to duplicate definitions for an identifier.
- It also hampers the readability of our code.

OUTPUT

```
SUM =  40
DIFFERENCE =  10
The value of MyVar =  345
```

Python Module Search Path (Locating Modules)

When a module is imported, Python interpreter searches for the requested module in several places. Firstly, it searches for a built-in module, if not found, then it starts searching into a list of directories defined in sys.path.

The search is in this order.

- The current directory.
- PYTHONPATH (an environment variable with a list of directory).
- If all else fails, Python checks the default path.

Note: On UNIX, this default path is normally /usr/local/lib/python

If we want to include our path where our modules are present, we can add or modify the path in the file “sys.path”.

Reloading a module

In a python program, we may include a module many times during a session, but The Python interpreter imports a module only once during a session. This makes things more efficient.

Example:

Create a module with the following content and save it as “testmodule.py”

```
<<testmodule.py>>

Print(“Hello, Welcome to Python Programming”);
Print(“I am from testmodule…….”;
```

Write the following 5 lines in a separate file and execute it.

```
import testmodule;
print("======== Good Morning ==========");
import testmodule;
print("========    Welcome      ==========");
import testmodule;
```

OUTPUT

```
Hello, Welcome to Python Programming
I am from testmodule....
======== Good Morning ==========
========    Welcome      ==========
```

Note: From the output it is clear that , even though the testmodule was imported many times, Python interpreter included it only once.

reload() method

Now if testmodule is modified during the course of the program, then, we would have to reload it.

One way to do this is to restart the interpreter. But this does not help much.

For this, Python provides the reload() function inside the *imp* module to reload a module.

Example:

```
import testmodule;
print("======== Good Morning ==========");
import testmodule;
print("========    Welcome   ==========");
import imp;
imp.reload(testmodule);
```

OUTPUT:

```
Hello, Welcome to Python Programming
I am from testmodule....
======== Good Morning ==========
========    Welcome   ==========
Hello, Welcome to Python Programming
I am from testmodule....
```

The dir() built-in function

dir() function is used to find out names that are defined inside a module.

It returns a sorted *list* of strings containing the names defined by a module.

Example:

```
import mymodule;
contents_of_mymodule = dir(mymodule);
print("Contents of mymodule are ");
i=int();
for i in contents_of_mymodule:
                print(i);
```

OUTPUT

```
Contents of mymodule are
MyVar
__builtins__
__cached__
__doc__
__file__
__loader__
__name__
__package__
__spec__
function1
function2
```

Note:

Here, we can see a sorted list of names.

All other names that begin with an underscore are default Python attributes associated with the module (we did not define them ourself).

For example, the `__name__` attribute contains the name of the module.

All the names defined in our current namespace can be found out using the `dir()` function without any arguments.

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

- 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.
- Therefore, 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. Then Python stops searching the local namespace for the variable.

Namespaces and Scoping - Example

Example:
For example, we define a variable X in the global namespace.
Within the function we assign *X a* value, therefore Python assumes *X* as a local variable.
However, we accessed the value of the local variable X before setting it,
so an UnboundLocalError is the result.

```
X = 2000;      # Glogal Variable
def Function1():
   X = X + 1;
   print("Value of X inside Function is = ",X);

   print("Value of X outside Function is = ",X);
# Python assumes X as a local variable,
# Since it is used before defining, it throws an error
"UnboundLocalError".

Function1();
```

OUTPUT

```
X = X + 1;
UnboundLocalError: local variable 'X' referenced before assignment
```

Namespaces and Scoping - Example

This error is fixed by placing the statement “global X” inside Function1()

```
X = 2000;      # Glogal Variable
def Function1():
  global X;
  X = X + 1;
  print("Value of X inside Function is = ",X);


  print("Value of X outside Function is = ",X);

# Python assumes X as a local variable,
#Since it is used before defining, it throws an error
"UnboundLocalError".

Function1();
```

OUTPUT
Value of X inside Function is = 2001
Value of X outside Function is = 2001

The *globals()* and *locals()* Functions

- The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.
- If locals() is called from within a function, it will return all the names that can be accessed locally from that function.
- If globals() is called from within a function, it will return all the names that can be accessed globally from that function.
- The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

The *globals()* and *locals()* Functions - Example

```
X = 2000;      # Glogal Variable
Y = 4000;      # Global Variable
def Function1():
   X = 200;    # Local Variable
   Y = 400;    # Local Variable

   print("========= List of Local elements =======");
   LocalElements = locals();
   print(LocalElements.keys());

   print("========= List of Global elements =======");
   GlobalElements = globals();
   print(GlobalElements.keys());

Function1();
```

OUTPUT

```
========= List of Local elements =======
dict_keys(['X', 'Y'])
========= List of Global elements =======
dict_keys(['__loader__', 'Function1', '__doc__', 'os', 'X', '__package__', 'Y',
'__builtins__', '__file__', '__name__', '__spec__', '__cached__'])
```

Packages in Python

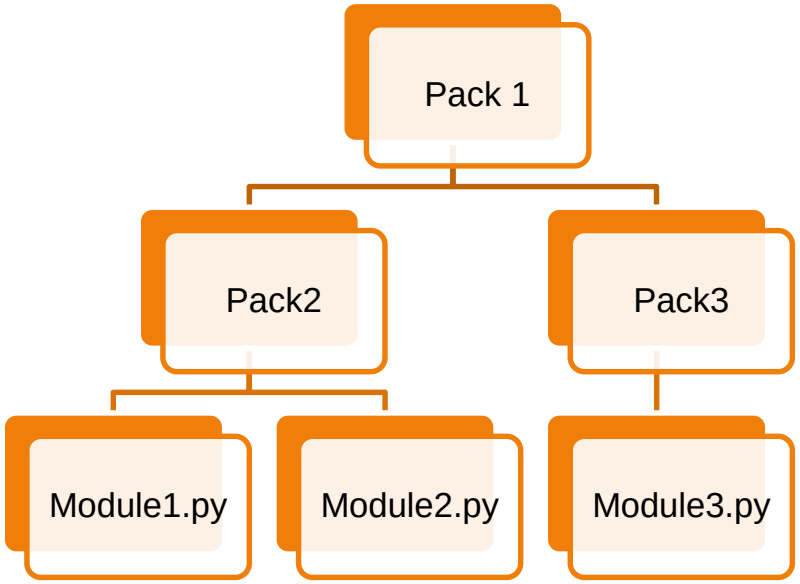
- Packages are nothing but directories or folders to stores files(modules).
- We normally don’t store all of our files in our computer in the same location.
- We use a well-organized hierarchy of directories for easier access.
- Similar files are kept in the same directory

Importing module from a package

Modules can be imported from packages using the dot (.) operator.

Example:

In the below hierarchy, If we want to import the module “Module3.py” which is in Package “Pack3” which is under Package “Pack1”, then we need to use the statement “import Pack1.Pack3.Module3”



Importing module from a package (Contd)

Now if this module Module3 contains a function named Function1(), then we must use the full name to reference it.

```
Import Pack1.Pack3.Module3.Function1()
```

If this construct seems lengthy, we can import the module without the package prefix as follows.

```
from Pack1.Pack3 import Module3
```

Then, We can call the function as follows.

```
Module3.Function1()
```

Importing module from a package (Contd)

Another way of importing just the required function (or class or variable) form a module within a package would be as follows.

```
from Pack1.Pack3.Module3 import Function1
```

Now we can directly call this function.

```
Function1();
```

Note:

Using the full namespace avoids confusion and prevents two same identifier names from colliding.

Questions