BCA – 502: Python Programming

Rahul Kumar Singh

In today's Class we have discussed on Python Modules.

What is a Module?

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code. Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application is known as Module.

Create a Module:

To create a module just save the code you want in a file with the file extension **.py**

Example:-

Here's an example of a simple module, **support.py**

```
def print_func( par ):
```

```
   print "Hello : ", par
   return 0
```

Use a Module:

Now we can use the module we just created, by using the import statement:

You can use any Python source file as a module by executing an import statement in some other Python source file.

The import has the following syntax –

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script –

Example:-

```
#!/usr/bin/python

# Import module support

import support

# Now you can call defined function that module as follows
```

```
support.print_func("Rahul")
```

Output:-

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

```
Hello : Rahul
```

Note: When using a function from a module, use the **syntax:** module_name.function_name.

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

Variables in Module:

The module can contain functions, as already described, but also variables of all types (lists, tupples, dictionaries, objects etc):

Example:- directory

Save this code in the file **myinfo.py**

```
person1 = {
  "name": "Rahul",
```

```
  "age": 29,
  "country": "India"
}
```

Import the module named myinfo, and access the person1 dictionary:

```
import myinfo
a = myinfo.person1["name"]
print(a)
```

Output:-

Rahul

The from...import Statement:

Python's from statement lets you import specific attributes from a module into the current namespace.

You can choose to import only parts from a module, by using the **from** keyword.

The from...import has the following syntax −

```
from modname import name1[, name2[, ... nameN]]
```

Example:-

The module named **myinfo** has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)
person1 = {
  "name": "Rahul",
  "age": 29,
  "country": "India"
}
```

Note :- save the above code as myinfo.py

Import only the person1 dictionary from the module:

```
from myinfo import person1
print (person1["country"])
```

Output:-

India

Note:- When importing using the **from** keyword, do not use the module name when referring to elements in the module.

Example:person1["country"], **not** ~~myinfo.person1["country"]~~

The from...import * Statement:

It is also possible to import all 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.

Re-naming a Module:

You can create an alias when you import a module, by using the **as** keyword:

Example:- Create an alias for myinfo called mi:

```
import myinfo as mi
a = mi.person1["age"]
print(a)
```

Output:-

29

Built-in Modules:

There are several built-in modules in Python, which you can import whenever you like.

Example:- Import and use the platform module:

```
import platform

x = platform.system()

print(x)
```

Output:-

```
Windows
```

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.

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. Python stops searching the local namespace for the variable.

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.

```
#!/usr/bin/python

Money = 2000

def AddMoney():

   # Uncomment the following line to fix the code:

   # global Money

   Money = Money + 1

print Money
```

AddMoney()

print Money

Output:- Error

Using 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 −

#!/usr/bin/python

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']

Here, the special string variable __name__ is the module's name, and __file__ is the filename from which the module was loaded.

Note: The dir() function can be used on all modules, also the ones you create yourself.

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