

## What is a Module?

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.

### In other words

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.

### Example

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

```
def print_func( par ):  
    print "Hello : ", par  
    return
```

## 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 –

```
#!/usr/bin/python  
  
# Import module support  
import support  
  
# Now you can call defined function that module as follows  
support.print_func("Zara")
```

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

Hello : Zara

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.

## The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax –

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

For example, to import the function `fibonacci` from the module `fib`, use the following statement –

```
from fib import fibonacci
```

This statement does not import the entire module `fib` into the current namespace; it just introduces the item `fibonacci` from the module `fib` into the global symbol table of the importing module.

## 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.

## 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
```

## The *reload()* Function

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The *reload()* function imports a previously imported module again. The syntax of the *reload()* function is this –

```
reload(module_name)
```

Here, *module\_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following –

```
reload(hello)
```

## Packages in Python

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

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

```
#!/usr/bin/python

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.

```
#!/usr/bin/python

# 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.