Modules

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. Modules present a whole group of functions, methods, or data that should relate to a common theme. To make a module usable, two things need to be available. First, the module itself has to be installed on the system. The simplest way to begin using a module is with the import keyword:

import sys

This will **import** the module named **sys** that contains services Python offers that mostly involve system - specific items.

A module can contain any Python code we like. All the programs we have written so far have been contained in a single .py file, and so they are modules as well as programs. The key difference is that ***programs are designed to be run, whereas modules are designed to be imported and used by programs***.

Not all modules have associated .py files—for example, the **sys** module is built into Python, and some modules are written in other languages (most commonly, C). However, much of Python's library is written in Python, so, for example, if we write ***import collections*** we can create named tuples by calling ***collections.namedtuple()***, and the functionality we are accessing is in the collections.py module file. It makes no difference to our programs what language a module is written in, since all modules are imported and used in the same way.

Several syntaxes can be used when importing. For example:

import *importable*
import *importable1*, *importable2*, ..., *importableN*
import *importable* as *preferred_name*
from *importable* import *

If we want to control exactly what is imported when the:

from module import *

syntax is used, we can define an **__all__** list in the module itself, in which case doing:

from module import *

will import only those objects named in the __all__ list.

For example, to import the **function fib** from the module **fib.py**, use the following statement-

```
# Fibonacci numbers module
def fib(n):
        ‘ return Fibonacci series up to n’
        result = []
        a, b = 0, 1
        while b < n:
                result.append(b)
                a, b = b, a+b
        return result
```

save the above code to fib.py

```
>>> from fib import fib
>>> fib(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

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

Executing Modules as Scripts

Within a module, **the module's name** (as a string) is available as the value of the global variable **__name__**.

The code in the module can be executed by executing the .py file itself , just as if you imported it, but with the __name__ set to "__main__".

Add this code at the end of your module-

```
# Fibonacci numbers module inside the fib.py file
def fib(n):             # return Fibonacci series up to n
        result = []
        a, b = 0, 1
        while b < n:
                result.append(b)
                a, b = b, a+b
        return result

if __name__ == "__main__":
        f=fib(100)
        print(f)
```

When you run the above code, the following output will be displayed.

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

The Python interpreter looks in the directories that are part of the **module search path**. These directories are listed in the **sys.path** variable from the **sys** module.

The path, or list of directories that Python should search through, is stored in the **sys** module, in a variable named **path**. To access this name, you will need to ***import the sys module***. Until you do that, the sys.path won't be available to you:

```
> > > import sys
> > > print(sys.path)
['C:/Python31/Chapter 6', 'C:\\Python30\\Lib\\idlelib',
'C:\\Windows\\system32\\python31.zip', 'C:\\Python31\\DLLs',
'C:\\Python31\\lib', 'C:\\Python31\\lib\\plat-win',
'C:\\Python31', 'C:\\Python31\\lib\\site-packages']
```

You can see that **sys.path** is a normal list, and if you want to add directories that will be checked for your modules, because you want them somewhere that isn't already in **sys.path**, you can alter it by using the usual methods — either the **append** method to add one directory, or the **extend** method to add any number of directories.

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.

Packages

A package is simply a directory that contains a set of modules and a file called **__init__.py.**

In some situations it is convenient to load in all of a package's modules using a single statement. To do this we must edit the package's **__init__.py** file to contain a statement which specifies which modules we want loaded. This statement must assign a list of module names to the special variable **__all__.**

For example, here is the necessary line for the Graphics/__init__.py file:

```
__all__ = ["Bmp", "Jpeg", "Png", "Tiff", "Xpm"]
```

That is all that is required, although we are free to put any other code we like in the **__init__.py** file. Now we can write a different kind of import statement:

```
from Graphics import *
image = Xpm.load("sleepy.xpm")
```

The ***from package import** **** syntax directly imports all the modules named in the **__all__** list.

So, after this import, not only is the Xpm module directly accessible, but so are all the others.

As noted earlier, this syntax can also be applied to a module, that is, ***from module import ****, in which case all the functions, variables, and other objects defined in the module (apart from those whose names begin with a leading underscore) will be imported