Module

A module is yet another python program file. The only requirement is that the filename without the extension should be an identifier – should start with a letter of English and should be followed by letters of English, digits or underscores.

Then why do we require a module?

- Share as a library
  If we write all our code in a single file, it cannot be used by others. So create the functions, classes in a separate file.
- Develop libraries in a domain
  This helps modular development. We develop all routines related to a particular domain. We have already used such modules like math, os and so on.
- Avoid name clashes
  The names used in a module do not normally clash with the names in another module.

```
# filename: ex1.py
# module:
#       physical unit of reuse
#       refers to a file
#       name should be an identifier
#       include using import statement
#       executed at the point during the program execution
#       not similar to #include of 'C' - executed before compilation
#       can be used to avoid clash of global names


print("one")
import module1
print("two")
print(module1, type(module1))
print("three")
```

```
# filename: module1.py
# all these statements are executed on import
print("hello from module1")
```

Let us consider the file module1.py.
As you can make out, it is just a regular python program.
Observe the following statement in client1.py.
import module1

We can execute the file module1.py in two ways.
1. directly as python module1.py
2. indirectly using import module1 in some other python program.

Import statement causes execution of the module dynamically when that point is reached during the execution of the client1.py file.
Import is not like #include of some other language, which literally copies the file.
Import does a few more things. We will explore all these as we go along.

Observe there is no change in the directory structure when we execute a python file directly. On import, the python file is compiled to an intermediate form and is stored in a directory called __pycache__ as a .pyc file.
Next time we import the same file in the same or any other program, the compiled code is directly used.
Observe using ls -l command the time of modification of .pyc file and the correspodning .py file. If the .py file is changed and is imported, a new .pyc file is created.
So, .pyc file is recreated only if required. This concept is called a build concept.

```
# filename: module 2
import math
def area_rect(l, b):
    return l * b

def area_triangle(a, b, c):
    s = (a + b + c) / 2.0
    area = math.sqrt(s * (s - a) * (s - b) * (s - c))
    return area

PI = 3.14
def area_circle(r):
    return PI * r * r
```

```
# filename : ex2.py

# can import # of modules using comma separated list
import module2
# find area of rectangle
#print("area : ", area_rect(20, 10)) # error

# should use a fully qualified name
print("area : ", module2.area_rect(20, 10))
print("area : ", module2.area_triangle(3, 4, 5))
print("area : ", module2.area_circle(7))
```

Let us now examine module2.py and client2.py.
The program file module2.py contains three functions to find areas of rectangle, triangle and circle. It also has a hard coded value of PI.
The user of the program client2.py imports the module module2.py.
He cannot access either functions or the variables of the module directly.
He can access them using the fully qualified name as modulename.functionname(<arg>).
There is no clash of names. The client can also have a variable called PI or a function called area_rect.

We shall discuss different ways of importing modules.

The code is taken from the file ex3_module2.py.

Import module2

This causes module2 to become an entity of type module.

We can access any member in module2 using fully qualified name.

There is no clash of names.

```
# different ways of import
# 1.
"""
import module2
print(module2, type(module2))
print("PI  ", module2.PI)
"""
```

To avoid using long module names, the user can alias them to a shorter name as in the following example.

import module2 as m2

From this point, m2 is an alias for module2 and any of them can be used interchangeably.

```
# 2. the user can create shorter and/or meaningful name
"""
import module2 as m2
print(m2, type(m2))
print("PI  ", m2.PI)
"""
```

To avoid using qualified long names, we may want to import a list of names from the module.

from module2 import PI, area_rect

In this case, the module name is no more available as a module type – so fully qualified name will not work. This way of import may cause name clashes.

```
# 3. selective import of symbols
"""
from module2 import PI, area_rect
# module2 is no more a Pythonic entity
#print(module2, type(module2))

# can access the members without qualifying
# cannot access those which are not imported
#print("area : ", module2.area_rect(20, 10)) # error
print("area : ", area_rect(20, 10))

#print("area : ", module2.area_circle(7)) # error
#print("area : ", area_circle(7)) # error
"""
```

A few questions for you to think.

```
# what if an imported function calls the other not imported function
# what if there is a function with the same name defined
#               before import
#               or after import
```

To avoid using qualified long names, we may want to import all names from a module to the current python program.

from module2 import *

* stands for all the names.

```
# 4. import all
from module2 import *
#print(module2, type(module2)) # error
print("area : ", area_rect(20, 10))
```

This is the least preferred way of importing.

Examine the files:

ex4_module3.py and module3.py.

These files ask you to follow a few steps to understand the build concept mentioned earlier.

We may execute the module directly or indirectly through the import mechanism. We may to do some testing when we execute directly which are not required when executed through the import mechanism.

How do we programmatically distinguish direct execution and execution through the import mechanism?

A module has a builtin variable whose name is __name__.

This variable has the name of the module on import and

has the value '__main__' on direct execution.

In the module, we use an idiom

if __name__ == '__main__':

and put the code to be executed on direct execution as the suite of the above selection.

```
# filename: module4.py

def foo():
        print("foo called")
def bar():
        print("bar called")

# will these be called when we
# a) execute this file directly : python module4.py
# b) import in another file : python client1_module4.py
# YES
# How do we avoid executing this test code on import?
# How to distinguish between direct execution and import
# check the variable:
#       __name__
#               on direct execution :  '__main__'
#               on import : 'module4' #module name
# code for testing
"""
# version 1
foo()
bar()
"""
# remove this statement later
print("__name__: ", __name__)
#version 2
if __name__ == '__main__':
        foo()
        bar()
```

```
# filename: ex_module4.py
# testing __name__
import module4
```

If we create the modules in our directories, we cannot easily share these files. Can we import modules if the files are elsewhere?
We can do that in two different ways.

```
# filename: MyLib/module5.py

# module5 in directory MyLib
def foo():
        print("hello from module5 of MyLib")
```

```
#filename: ex5_MyLib_module5.py
```

```
#import module5 # error
#module5.foo()

#import MyLib.module5 # error
#module5.foo()

import sys
print(sys.path)
sys.path.insert(0, '/home/nsk/pes/ICUP/Theory/module/MyLib/')
print(sys.path)
import module5
module5.foo()
```

---

import checks for the filename in the current directory as well as all directories whose names are stored in a list variable sys.path.
To import modules from some other directory, insert the path of the module in the variable sys.path.

It is also possible to change a special environmental variable PYTHONPATH in the operating system.
In unix flavours, we can do this as follows.

```
$ export PYTHONPATH=/home/nsk/pes/ICUP/Theory/module/MyLib:$PYTHONPATH
```

Thats about modules.