What are modules in Python?

Modules refer to a file containing Python statements and definitions.
A file containing Python code, for e.g.: example.py, is called a module and its module name would be example.
We use modules to break down large programs into small manageable and organized files.
Furthermore, modules provide reusability of code.
We can define our most used functions in a module and import it, instead of copying their definitions into different programs.
Let us create a module. Type the following and save it as example.py.

```
# Python Module example

def add(a, b):
   """This program adds two
   numbers and return the result"""

   result = a + b
   return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

How to import modules in Python?
We can import the definitions inside a module to another module or the interactive interpreter in Python.
We use the import keyword to do this. To import our previously defined module example we type the following in the Python prompt.

```
>>> import example
```

This does not enter the names of the functions defined in example directly in the current symbol table. It only enters the module name example there.
Using the module name we can access the function using the dot . operator. For example:

```
>>> example.add(4,5.5)
9.5
```

Python has a ton of standard modules available.
You can check out the full list of Python standard modules and what they are for. These files are in the Lib directory inside the location where you installed Python.
Standard modules can be imported the same way as we import our user-defined modules.
There are various ways to import modules. They are listed as follows.
Python import statement
We can import a module using import statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
# import statement example
# to import standard module math
import math
print("The value of pi is", math.pi)
```

Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module then (if not found) 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).
The installation-dependent default directory.

```
>>> import sys
>>> sys.path
['',
'C:\\Python33\\Lib\\idlelib',
'C:\\Windows\\system32\\python33.zip',
'C:\\Python33\\DLLs',
'C:\\Python33\\lib',
'C:\\Python33',
'C:\\Python33\\lib\\site-packages']
```

We can add modify this list to add our own path.

Reloading a module
The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works.
Suppose we have the following code in a module named my_module.

```
# This module shows the effect of
#  multiple imports and reload

print("This code got executed")
```

Now we see the effect of multiple imports.

```
>>> import my_module
This code got executed
>>> import my_module
>>> import my_module
```

We can see that our code got executed only once. This goes to say that our module was imported only once.
Now if our module changed during the course of the program, we would have to reload it.One way to do this is to restart the interpreter. But this does not help much.
Python provides a neat way of doing this. We can use the reload() function inside the imp module to reload a module. This is how its done.

```
>>> import imp
>>> import my_module
This code got executed
>>> import my_module
>>> imp.reload(my_module)
This code got executed
<module 'my_module' from '.\\my_module.py'>
```

The dir() built-in function

We can use the dir() function to find out names that are defined inside a module.

For example, we have defined a function add() in the module example that we had in the beginning.

```
>>> dir(example)
['__builtins__',
'__cached__',
'__doc__',
'__file__',
'__initializing__',
'__loader__',
'__name__',
'__package__',
'add']
```

Here, we can see a sorted list of names (along with add). 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.

```
>>> import example
>>> example.__name__
'example'
```

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

```
>>> a = 1
>>> b = "hello"
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'b', 'math', 'pyscripter']
```

What are packages?

We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access.

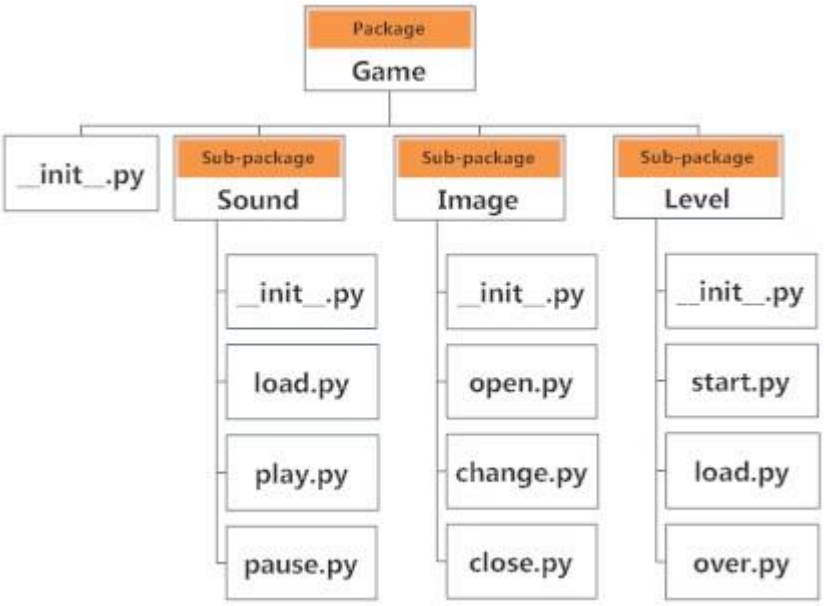
Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.

A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.



Importing module from a package

We can import modules from packages using the dot (.) operator.

For example, if want to import the `start` module in the above example, it is done as follows.

```
1. import Game.Level.start
```

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

```
1. Game.Level.start.select_difficulty(2)
```

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

```
1. from Game.Level import start
```

We can now call the function simply as follows.

```
1. start.select_difficulty(2)
```

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

```
1. from Game.Level.start import select_difficulty
```

Now we can directly call this function.

```
1. select_difficulty(2)
```

Although easier, this method is not recommended. Using the full namespace avoids confusion and prevents two same identifier names from colliding.

While importing packages, Python looks in the list of directories defined in `sys.path`, similar as for module search path.