Package in Python

What are python packages?

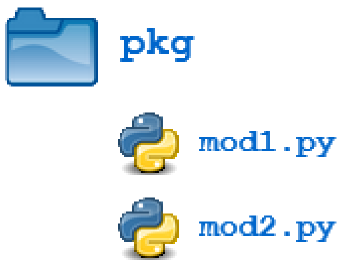
We don't usually store all of our files in our computer in the same location. We use a well-organized **hierarchy** of directories for easier access.

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.

Packages allow for a hierarchical structuring of the module namespace using **dot notation**. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.



Here, there is a directory named pkg that contains two modules.

The contents of the modules are:

mod1.py

```
def foo():
        print('[mod1] foo()')
class Foo:
        pass
```

mod2.py

```
def bar():
        print('[mod2] bar()')
class Bar:
        pass
```

You can refer to the two **modules** with **dot notation** (pkg.mod1, pkg.mod2) and import them with the syntax you are already familiar with:

```
import <module_name>[, <module_name> ...]
```

You can import modules with these statements as well:

```
from <package_name> import <modules_name>[, <module_name> ...]from <package_name> import <module_name> as <alt_name>
```

But this is of little avail. Though this is, strictly speaking, a syntactically correct Python statement, it doesn't do much of anything useful.

In particular, it does not place any of the modules in pkg into the local namespace:

```
>>> pkg.mod1Traceback (most recent call last):  File "<pyshell#34>", line 1, in <module>    pkg.mod1AttributeError: module 'pkg' has no attribute 'mod1'
```

```
>>> pkg.mod1.foo()

Traceback (most recent call last):

File "<pyshell#35>", line 1, in <module>

    pkg.mod1.foo()

AttributeError: module 'pkg' has no attribute 'mod1'
```

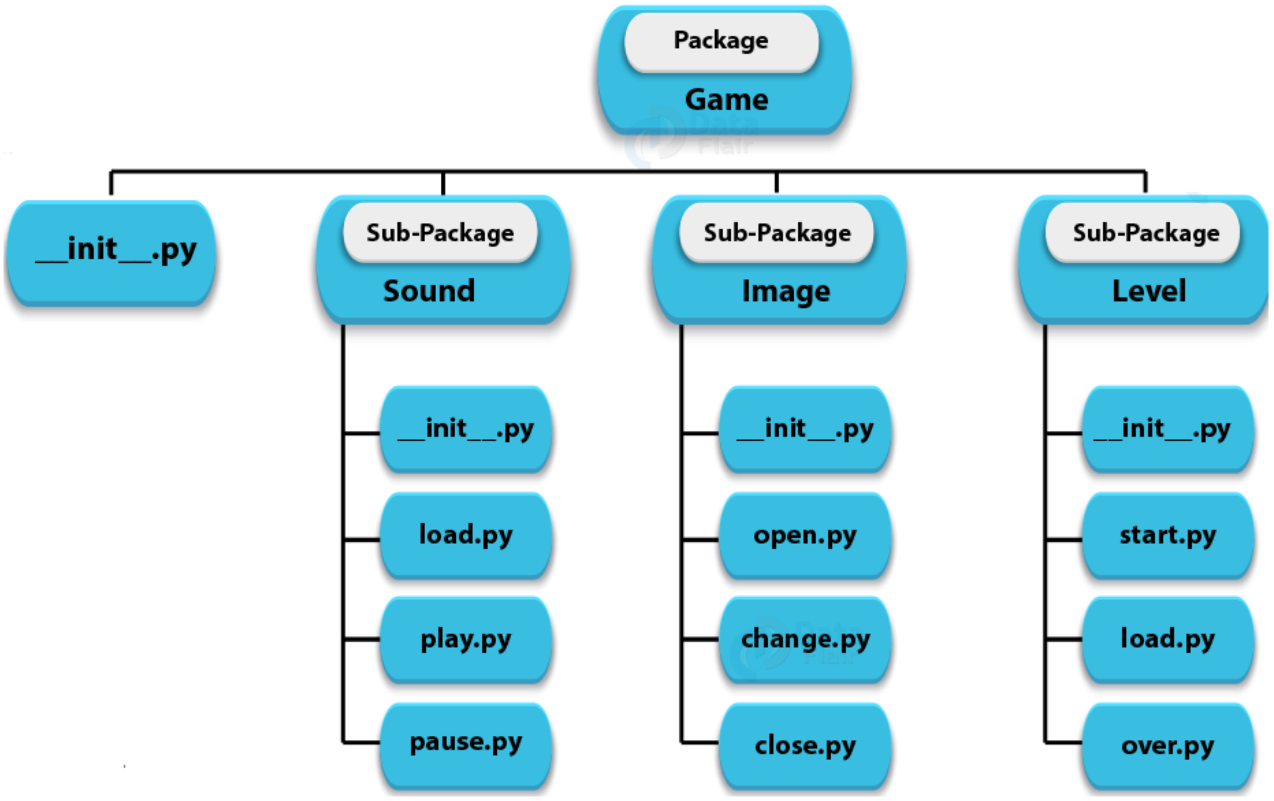
To actually import the modules or their contents, you need to use one of the forms shown above.

If a file named __init__.py is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

NOTE:

*Much of the Python documentation states that an __init__.py file **must** be present in the package directory when creating a package. This was once true. It used to be that the very presence of __init__.py signified to Python that a package was being defined. The file could contain initialization code or even be empty, but it **had** to be present.*

*Starting with **Python 3.3**, Implicit Namespace Packages were introduced. These allow for the creation of a package without any __init__.py file. Of course, it **can** still be present if package initialization is needed. But it is no longer required.*



Example:

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.