Python Packages

Python Modules: Overview

- Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules.
  - Simplicity
  - Maintainability
  - Reusability
- A module's contents are accessed with the import statement.

The Module Search Path

- When the interpreter executes the import statement, it searches in a list of directories assembled from the following sources:
  - The current directory
  - The list of directories contained in the PYTHONPATH environment variable
  - An installation-dependent directory

    - >>> import sys
    - >>> sys.path
    - ['', 'C:\\Users\\john\\Documents\\Python\\doc', 'C:\\Python36\\Lib\\idlelib',
    - 'C:\\Python36\\python36.zip', 'C:\\Python36\\DLLs', 'C:\\Python36\\lib',
    - 'C:\\Python36', 'C:\\Python36\\lib\\site-packages']

The import Statement

- import <module_name>
- from <module_name> import <name(s)>
- from <module_name> import <name> as <alt_name>
- import <module_name> as <alt_name>

The dir() Function

- The built-in function dir() returns a list of defined names in a namespace.

  - >>> dir()
  - ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
  - '__package__', '__spec__']

  - >>> qux = [1, 2, 3, 4, 5]
  - >>> dir()
  - ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
  - '__package__', '__spec__', 'qux']

Executing a Module as a Script

- Any .py file that contains a module is essentially also a Python script, and there isn't any reason it can't be executed like one.

__main__

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]
def foo(arg):
    print(f'arg = {arg}')
class Foo:
    pass
if (__name__ == '__main__'):
    print('Executing as standalone script')
    print(s)
    print(a)
    foo('quux')
    x = Foo()
    print(x)
```

Reloading a Module

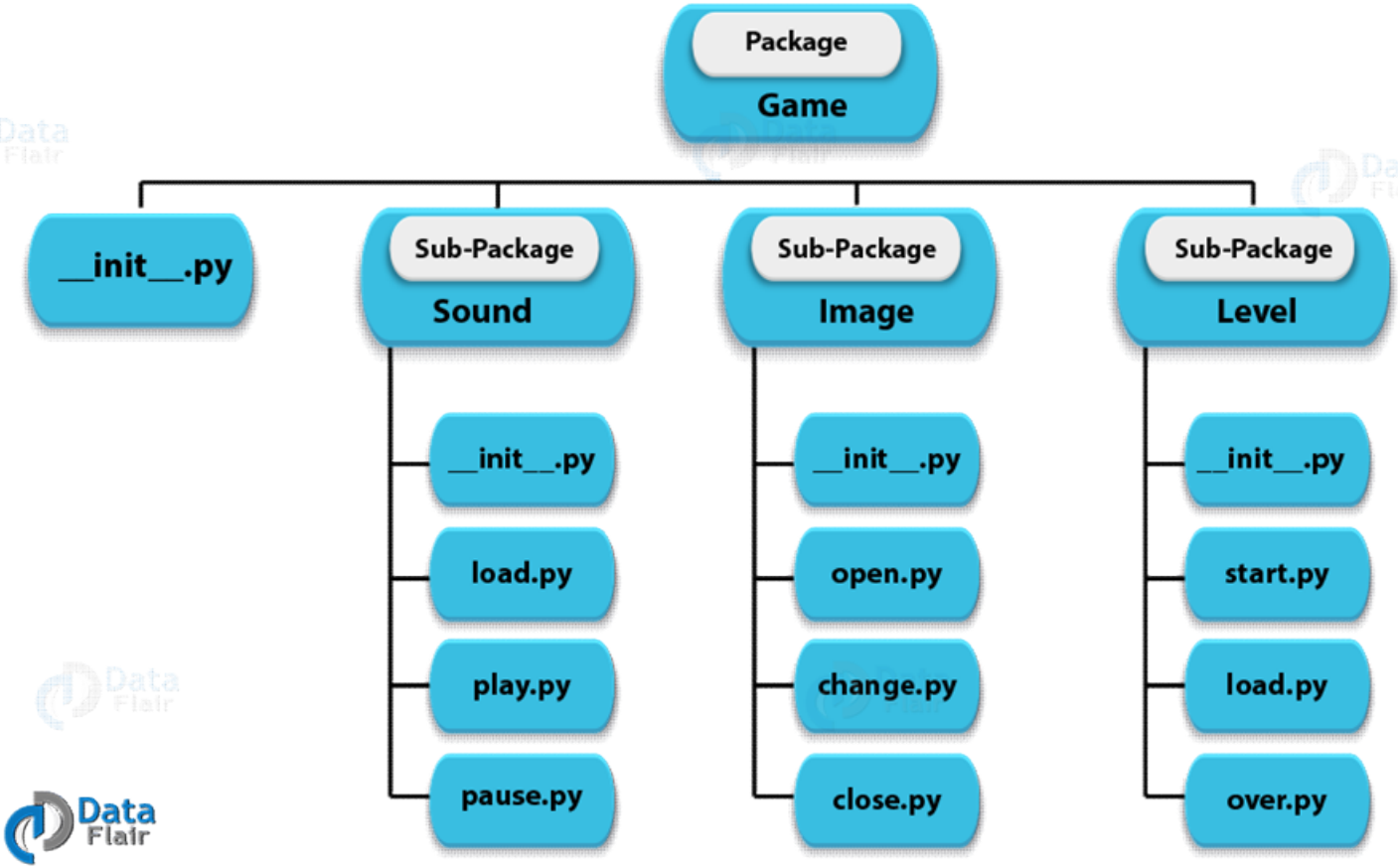
- For reasons of efficiency, a module is only loaded once per interpreter session.
- A module can contain executable statements as well, usually for initialization.
- Be aware that these statements will only be executed the first time a module is imported.

    - >>> import importlib
    - >>> importlib.reload(mod)

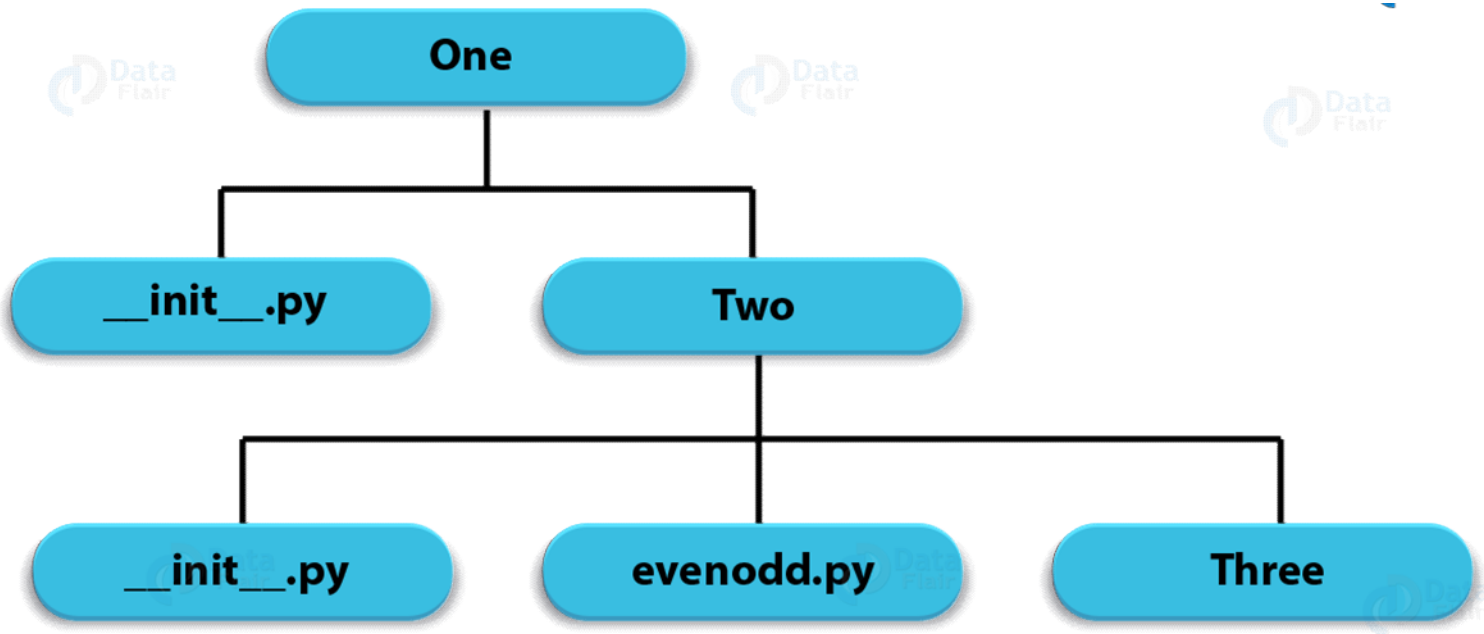
Python Packages

- Suppose you have developed a very large application that includes many modules.
- As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location.
- Packages allow for a hierarchical structuring of the module namespace using dot notation.
- Creating a package is quite straightforward

Package Module Structure



How to Create Your Own Python Package?



Import Packages

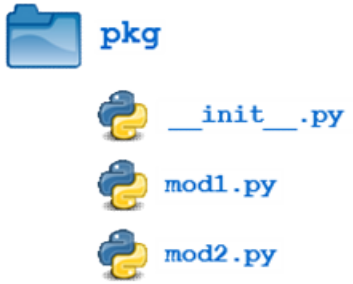
- import pkg.mod1, pkg.mod2
  - pkg.mod1.foo()
- from pkg.mod1 import foo
  - foo()
- from pkg import mod1
  - mod1.foo()
- import pkg
  - pkg.mod1.foo()
  - Traceback (most recent call last):
  - File "<pyshell#35>", line 1, in <module>
  - pkg.mod1.foo()
  - AttributeError: module 'pkg' has no attribute 'mod1'

Package Initialization

- 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.

__init__.py

- print(f'Invoking __init__.py for {__name__}')
- A = ['quux', 'corge', 'grault']



Package Initialization

- then when you execute import pkg, modules mod1 and mod2 are imported automatically:
  - >>> import pkg
  - Invoking __init__.py for pkg
  - >>> pkg.mod1.foo()
  - [mod1] foo()
  - >>> pkg.mod2.bar()
  - [mod2] bar()

Subpackages

- Packages can contain nested subpackages to arbitrary depth
  - >>> import pkg.sub_pkg1.mod1
  - >>> pkg.sub_pkg1.mod1.foo()
  - [mod1] foo()
  - >>> from pkg.sub_pkg1 import mod2
  - >>> mod2.bar()
  - [mod2] bar()
  - >>> from pkg.sub_pkg2.mod3 import baz
  - >>> baz()
  - [mod3] baz()
  - >>> from pkg.sub_pkg2.mod4 import qux as grault
  - >>> grault()
  - [mod4] qux()

Thanks