Persistent

Modules and Packages

Objectives

At the end of this session, you will be able to understand:

- Python Modules/Packages

Persistent

Modules/Packages

Python module

- A module is a file containing code
- A module defines a group of Python functions or other objects, and the name of the module is derived from the name of the file
- Modules allow a group of related, yet independently operating functions to be shared to take advantage of work that has been done, maximizing code reusability
- In a nutshell, Python Modules are self-contained and organized pieces of Python code that can be shared and reused

A first Python module

- Create a file called mymath.py and enter the following Python code in the file

```
"""mymath – our example math module"""

pi = 3.14159

def area(r):

    """area(r): return the area of a circle with radius r."""

    global pi

    return(pi * r * r)
```

- Save the file. As with functions, modules also provide the option of putting in a document string as the first line of the module

Persistent

A first Python module (contd.)

- Start Python shell and type the following

```
>>> pi
Traceback (innermost last):
File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> area(2)
Traceback (innermost last):
File "<stdin>", line 1, in ?
NameError: name 'area' is not defined
```

In other words, Python doesn’t have the constant **pi** or the function **area** built in.

A first Python module (contd.)

- Now type the following

```
>>> import mymath
>>> pi
Traceback (innermost last):
File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> mymath.pi
3.1415899999999999
>>> mymath.area(2)
12.56636
```

The module import statement

- The module *import* statement takes three different forms.
- The most basic form is *import modulename.* This form searches for a Python module of the given name, parses its contents, and makes it available. The importing code can use the contents of the module, but any references by that code to names within the module must still be prefixed with the module name. If the named module isn't found, an error will be generated.
- The second form is *from modulename import name1, name2, name3, . . .*. This form permits specific names from a module to be explicitly imported into the code. Each of name1, name2, and so forth from within modulename is made available to the importing code; code after the import statement can use any of name1, name2, name3, and so on without prefixing the module name.
- Finally, there's a general form of *from modulename import * .* The * stands for all the exported names in modulename. This imports all public names from modulename - that is, those that don't begin with an underscore, and makes them available to the importing code without the necessity of prefixing the module name.

Importing modules

- Use classes & functions defined in another file
- A Python module is a file with the same name (plus the **.py** extension)
- Like Java **import**, C++ **include**

- Few formats of the command:

  import somefile

  import somefile as a

  from somefile **import** *

  from somefile **import** className

Persistent

Import…

- **import** somefile

  somefile.className.method(“**abc**”)

  somefile.myFunction(34)

- **import** somefile as a

  a.myFunction(34)

- **from** somefile **import** *

  className.method(“**abc**”)

  myFunction(34)

- **from** somefile **import** className

  className.method(“**abc**”) ← imported

  myFunction(34) ← Not imported

Executing modules as scripts

Example: Create a file called fibo.py with the below code content

```
# Fibonacci numbers module

def fib(n): # write Fibonacci series up to n

a, b = 0, 1

while b < n:

    print b

    a, b = b, a+b
```

- When a Python module is run with **python fibo.py <arguments>** the code in the module will be executed, just as if it were imported, but with the **__name__ set to "__main__"**.
- That means that the file can be made usable as a script as well as an imported module by adding the following code at the end of the module because the code that parses the command line only runs if the module is executed as the “main” file:

```
if __name__ == "__main__":

    import sys

    fib(int(sys.argv[1]))
```


Persistent

“Compiled” Python files

- As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called fibo.pyc exists in the directory where fibo.py is found, this is assumed to contain an already-“byte-compiled” version of the module fibo. The modification time of the version of fibo.py used to create fibo.pyc is recorded in fibo.pyc, and the .pyc file is ignored if these don’t match
- Whenever fibo.py is successfully compiled, an attempt is made to write the compiled version to fibo.pyc. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting fico.pyc file will be recognized as invalid and thus ignored later. The contents of the fibo.pyc file are platform independent, so a Python module directory can be shared by machines of different architectures
- When the Python interpreter is invoked with the –o flag, optimized code is generated and stored in .pyo files. The optimizer currently doesn’t help much; it only removes assert statements. When –o is used, all bytecode is optimized; .pyc files are ignored and .py files are compiled to optimized bytecode

Directories for module files

- Where does Python look for module files?
- The list of directories where Python will look for the files to be imported is sys.path
- This is just a variable named ‘path’ stored inside the ‘sys’ module

  >>> import sys

  >>> sys.path

  [‘’, '/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/site-packages/setuptools-0.6c5-py2.5.egg’, …]

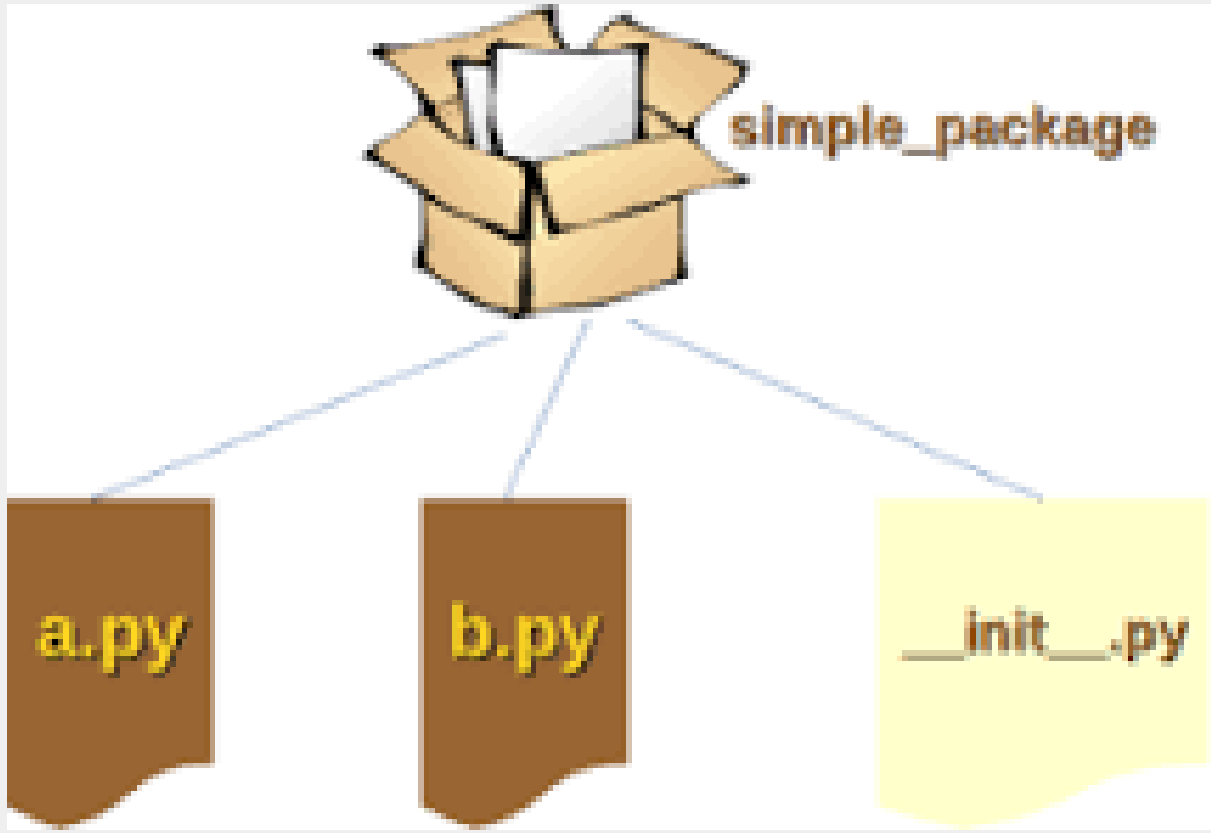
- To add a directory of your own to this list, append it to this list

  sys.path.append(‘/my/new/path’)

Persistent

Python package

- A **package** is basically a directory with **Python** files and a file with the name __init__.py. This means that every directory inside of the **Python** path, which contains a file named __init__.py, will be treated as a **package** by **Python**. It's possible to put several modules into a **Package**.



Persistent

Python package Example

- First of all, we need a directory. The name of this directory will be the name of the package, which we want to create. We will call our package **"simple_package"**. This directory needs to contain a file with the name **"__init__.py"**. This file can be empty, or it can contain valid Python code. This code will be executed when a package will be imported, so it can be used to initialize a package
- We create two simple files a.py and b.py just for the sake of filling the package with modules.

- Content of **a.py:**

```
def bar():
    print("Hello, function 'bar' from module 'a' calling")
```

- The content of **b.py:**

```
def foo():
    print("Hello, function 'foo' from module 'b' calling")
```

Python package Example

- Content of **__init__.py:**

  import simple_package.a

  import simple_package.b

- **Now try package access as:**

  >>> import simple_package

  >>> simple_package.a.bar()

  Hello, function 'bar' from module 'a' calling

  >>> simple_package.b.foo()

  Hello, function 'foo' from module 'b' calling

Some built-in Modules

sys module

- **Command Line Arguments**

- Common utility scripts often need to process command line arguments. These arguments are stored in the sys module’s argv attribute as a list. For instance the following output results from running -

  python demo.py one two three at the command line:

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

os module

- The **os** module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()                    # Return the current working directory 'C:\\Python27'
>>> os.chdir('/server/accesslogs’)  # Change current working directory
>>> os.system('mkdir today')       # Run the command mkdir in the system shell
```

```
>>> dir(os)
<returns a list of all module functions>
```

```
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Summary:

With this we have come to an end of this session, where we discussed about….

- Python Modules/Packages

In the next session we will discuss about

- File Handling

Persistent

Reference material

- http://www.tutorialspoint.com/python
- http://www.learnpython.org/
- http://docs.python.org/2/tutorial/
- https://docs.python.org/2/tutorial/stdlib.html
- https://docs.python.org/2/tutorial/stdlib2.html
- https://packaging.python.org/installing/

Questions

Key contacts

Sakshi Jamgaonkar

sakshi_jamgaonkar@persistent.com

Asif Immanad

asif_immanad@persistent.co.in

Persistent

Persistent

Thank you!