POWERING POTENTIAL

Modules and Packages

Module 15

MODULES AND PACKAGES

Contents

- What are modules and packages?
- How does Python find a module?
- Multiple source files
- Importing a module
- Importing names

What are modules?

- **A module is a file containing code**
  - Usually, but not exclusively, written in Python
  - Usually with a .py filename suffix (some modules are built-in)
- **A module might be byte-code**
  - Python will create a .pyc file if none exists
  - Held in subdirectory `__pycache__` from Python 3.2
  - Python will overwrite this if the .py file is younger
- **A module might be a DLL or shared object**
  - With a .pyd filename suffix
  - Often written in C as a Python extension

"Modules should have short, all-lowercase names"

PEP008

What are packages?

- **A package is a logical group of modules**
- **A directory containing a set of modules is a package**
- **The difference is a file called `__init__.py`**
  - Often empty
  - Can contain initialisation code
  - Can even contain functions
  - Can contain a list of the public interfaces as attribute `__all__`

→ These are the names imported with `from` *Module* `import *`

```
# Public interface
__all__ = ['getprocs', 'getprocsall', 'filter']
```

- See *Namespace packages* later…

Multiple source files – why bother?

- **Increase maintainability**
  - Independent modules can be understood easily
- **Functional decomposition**
  - Simplify the implementation
- **Encapsulation & information hiding**
  - Easier re-use of modules in a different program
  - Easier to change module without affecting the entire program
- **Support concurrent development**
  - Multiple people working simultaneously
  - Debug separately in discrete units
- **Promote reuse**
  - Logical variable and function names can safely be reused
  - Use or adapt available standard modules

How does Python find a module?

- **The initial path is from sys.path**
  - May be modified using `sys.path.append(`*dirname*`)`
  - Starts with the directory from which the main program was loaded

```
import sys
sys.path.append('./demomodules')
import mymodule
print(sys.path)
```

```
['C:\\QA\\Python\\MyDemos', 'C:\\Python30\\Lib', ...
./demomodules]
```

- **Or change environment variable PYTHONPATH**
  - Contains a list of directories to be searched
  - Separator is the same as your system's PATH

→ **:** for *NIX **;** for Windows

Importing a module

Surprisingly, use the import command

- At the top of your program, by convention

```
import mymodule
print(mymodule.attribute)
```

Case sensitive, even on Windows

- Can specify a comma-separated list of module names

```
import mymodule_a, mymodule_b, mymodule_c
```

- Can specify an alias for a module name

```
import mymodule_win32 as mymodule
print(mymodule.attribute)
```

- Trouble is, you have to specify the module name for each call

Importing names

- **Alternatively, import the names into your namespace**

```
from mymodule import *
```

- Beware! Risk of name collisions!

- **Specify specific object name(s)**

```
from mymodule import my_func1
...
my_func1()
```

How do we know which module my_func1 came from?

- **Or use an alias**

```
from mymodule import \
     (my_func1 as mf1, my_func2 as mf2)

mf1()
mf2()
```

Better or worse?

The 'main' trick

- **Code outside of a function is executed at import time**
  - That is undesirable if our module could be run as a program
- **Fortunately, we can test the name of the module**
  - Will be __main__ if run as a program

```
def main():
    """
    Stand-alone program code,
    usually function calls or tests
    """

if __name__ == "__main__":
    main()
```

Now our code can be run as a module or a stand-alone program

- Using a function called `main()` is not mandatory, but common practice

SUMMARY

- **Writing a module in Python is simple**
- Just a bunch of code in a file
- **Python loads modules based on sys.path**
- **Import a module using import**
- Can also specify importing names into our namespace