POWERING POTENTIAL

Python 3 Programming

Modules and Packages

MODULES AND PACKAGES

Contents

We have already met modules that are bundled with Python, now we shall discuss writing our own.

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

The modules we have seen so far have been bundled with Python, and we have used them as just another part of the language. The built-ins are not physically separate modules, although we have a logical view of them as being just like any other. Those aside, all modules are represented by separate files, which are logically independent from the main program.

Rather than compile a module each time it is loaded, Python compiles once and dumps the byte-code into another file. From Python 3.2, these files are held in a sub-directory called `__pycache__`.

Byte-code files (.pyc) are not necessarily portable, either across platforms or between releases. The files themselves contain a "magic number" which indicates the Python release they were built for, and incompatible byte-code files will be recreated. It is therefore important that a bundled package includes the source code (.py files) as well as the byte-code files if required. This has lead to issues where multiple versions of Python are regularly used, in that Python would be continually recompiling. From Python 3.2, the version of Python is also included in the filename in `__pycache__`, for example: `abc.cpython-32.pyc`.

Modules written as C extensions are generally created as a DLL (on Windows) or shared object (Linux/UNIX .so files). These have the file extension .pyd, but are otherwise identical to a native binary.

Prior to Python 3.5 we also had .pyo files. The .pyo files contained optimised byte-code and were created using the -O option to the python command-line. See PEP 488.

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…

In Python, a module is the file itself, and a package is a group of modules in a directory (or folder, if you prefer). The directory itself is the package - provided it has a file called **`__init__.py`** in it .*
This file is often empty, or maybe just has a comment in it. It can also have a huge amount of code in it, depending on the whim of the author. One of the more useful attributes which can optional be set is **`__all__`**, which gives a list of the public elements of the package.

* At least, that is the situation in Python 2, and up to Python 3.2. From Python 3.3, we have *Namespace packages* that don't have this file. Namespace packages are discussed 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

A Python module is somewhat like a separate source file or DLL in C or C++.
However, it is more than that:

Variables can be local to the module
Packages have an independent namespace
For OO-programming, each module can implement a single class

The term *package* is used to indicate a collection of modules, on a local disk or stored on the network. Specifically in Python, it is the directory that the modules reside in.
The reasons for splitting-up an application into modules are all based on good structured programming techniques, however, the overriding reason is code reuse - why reinvent when someone else has already written the code.

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

The directories searched for Python modules will vary depending on the platform and installation, but always includes the current directory. Windows also has C:\Python3*n*\Lib\site-packages. To find the path used just print **`sys.path`**.
To add a directory to the path, either use **`sys.path.append`**, or the environment variable **`PYTHONPATH`**.
Note that either directory separator (/ or \\) may be used on 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

We have seen the basics of importing modules already - after all, you can't do much in a Python program without using **`import`**. Here are a few more details.

Notice that the case of the module name must match that of the file name. By default, this also applies to Microsoft Windows, unless the environment variable `PYTHONCASEOK` is set. We can specify an alias if required, and that is a commonly used feature.

Modules already loaded can be reloaded using (surprise) **`imp.reload`**. This may be useful if you are creating the modules programmatically.

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?

Specifying the name of the module for each function call can be tedious, so we can import all the external names on the module into our own namespace. The problem is that this can lead to "Namespace pollution", so instead, we can specify exactly which names to import.
If those names clash with existing names within the program (*name collisions*), then we can assign aliases to individual names. However, it can then get very difficult to track back which names belong to which module, so choose your aliases carefully!
Note that we can only import public names, that is those not prefixed with an underscore, or those specified in **`__all__`**.

Directories as packages

- **Keep related modules together in the same directory**
  - The name should not be the same as a Python system directory
- **An `__init__.py` file is required**
  - Might be empty

Directory name/package name

```
import workingmodules.mymodule_a

workingmodules.mymodule_a.myfunc1()
```

- **May be nested**
  - Each nested sub-directory should have a `__init__.py` file
  - Each is just another name in the hierarchy
  - Import relative to the current package using *.module*

→ Import relative to the parent using *..module*

A *package* is a group of logically associated modules, and is useful for organising and distributing complex groups. Only the top level directory (the parent to the package) need be in the search path. The **`__init__.py`** file is a required file in the package directory, but it can be empty. It can contain initialisation code, and the definition of attribute **`__all__`**, which is a list of symbols to be exported when **`import *`** is used. Such a directory is called a *Regular package*, the alternative is discussed on the next slide.
Package directory names, like module file names, must conform to Python's naming rules, that is may consist of alpha-numerics or underscores but may not start with a numeric.

Namespace packages (3.3)

From Python 3.3 `__init__.py` is no longer mandatory

- A directory without `__init__.py` is a *Namespace package*
- A directory with `__init__.py` is a *Regular package*

Advantages:

- We no longer need to supply an empty `__init__.py`
- Namespaces can now span directories

```
sys.path.append('./date_packages')
sys.path.append('./person_packages')
from mynames.date import Date
from mynames.person import Person
```

Where both directories have a sub-directory named **`mynames`**

Disadvantages:

- No initialisation code
- No __name__ attribute for the namespace

py3

If all you want a package for is to logically group modules together, then the traditional package mechanism (*Regular packages*) is rather inflexible. The initialisation performed by `__init__.py` can be very powerful, but it is not always appropriate.
Enter *Namespace packages* at Python 3.3. These allow a subdirectory to be the namespace, but that same subdirectory name can occur in any number of other parent directories. It is the subdirectory name that is used for the Namespace. See PEP0420 for further details.

Writing a module

- **No special header or footer required in the file**
  - Just write your code without a 'main'
  - Default documentation is generated and available through help()
- **Conventions with underscores - reminder**
  - Names beginning with one underscore are private to a module

  → Includes function names
  - Names beginning and ending with two underscores have a special meaning
- **Name of the module is available in `__name__`**

```
def my_func1():
    print("Hello from", __name__)
```

A module in Python required no special header or delimiter in the file - any Python code file can be a module. The module can be without a "main", or a #! line and execute access (on UNIX), but see later when we discuss testing.
A single underscore prefix means that the name is not exported from a module, unless **`__all__`** is specified in the package **`__init__.py`** file, in which case only those names will be exported. Names with two leading underscores are mangled, and so localised.

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

It is not uncommon to develop a Python script and then realise that the majority of it would be useful as a module. Of course that requires that we have written it by breaking down the functionality into callable functions, which good programmers will do naturally anyway. In a full program, there is always the need for a 'main', which might do nothing more than call functions in the correct order, but that would get run at import time if we tried to run our program as a module.

We could just remove the 'main' code, but that would make life more difficult if we wanted to have the choice of running it as a module or a program. So, we can use trickery by testing the module name. We don't know what you will choose as your module name, but it won't/can't be `__main__`. That is the name used for the main module when running a program. So we can test the module attribute `__name__` and choose which code to execute.

It is common practice to use a function called `main`, since that gives us the opportunity to have scoped variables (remember that in Python a conditional statement is not a unit of scope).

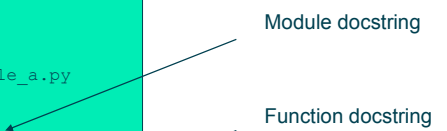
Some advocate *always* writing Python code in this way, including stand-alone programs, for maximum flexibility.

Module documentation

Docstring for the module must be at the (very) start

- Or explicitly assigned to __doc__
- Used by the pydoc utility to generate documentation files
- A default help format is provided

```
>>> help(mymodule_a)
Help on module mymodule_a:

NAME
    mymodule_a

FILE
    c:\qa\python\mydemos\demomodules\mymodule_a.py

DESCRIPTION
    This is a test module containing one
    function, my_func1

FUNCTIONS
    my_func1()
        my_func1 has no parameters and prints 'Hello'

DATA
    var1 = 42
```



Like functions, modules can contain docstrings, and these will be used as the documentation when **`help()`** is called. The format of the docstring is the same as for functions, and must occur at the very start of the module, even before any imports. If not at the start of the module, it can be assigned to the variable `__doc__`, for example:

```
__doc__ = """
        This is a sample module which
        does various date operations.
        """
```

Documentation in forms other than help() can be generated, using the **`lib/pydoc.py`** program (bundled with python). Documentation in HTML and UNIX man page format can be created, as well as searched using a small browser.
This is very useful on its own, but docstrings have other hidden magic...

Testing a module

Run a module as a main program

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Testing is built into Python

- Searches for docstrings containing interactive sessions

```
"""
    This is a sample module which
    does various date operations.

    >>> today = Date(13,12,1949)
    >>> print(today)
    13/12/1949
"""
```

Run with the -v option

```
$ date.py -v
Trying:
    today = Date(13,12,1949)
Expecting nothing
ok
Trying:
    print (today)
Expecting:
    13/12/1949
ok
...
Test passed.
```

We can write a test harness for our modules, that is a main program which calls the module, exercising each function. But there are problems with this approach. In the passage of time, the test harness gets lost, or does not get delivered with the module. Modifications done to the module are often omitted from the test harness, and it gets out of step.

A better strategy is to embed the tests inside the module - that way there is no extra test harness file to go missing, and the tests can be seen by anyone altering the code. Running the module as a main program will run the tests if we use the standard **`doctest`** module. The **`doctest`** module searches for docstrings and, in its simplest form, will just check these. If an interactive session is found within a docstring, then it will be seen as a test an run. It looks for the >>> prompt as a command with output which follows.

The test for `__name__` will only be true if the module is run as a program, not if it is imported. You don't have to use **`doctest`**, you could write your own tests, but that is like reinventing the wheel.

Note that if you execute the module as main then you will need a #! line and execute access on UNIX.

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
- **Directories can be packages**
- Require the `__init__.py` file
- **Python supports module documentation**
- docstrings
- **There are several features and base modules to assist testing**

Distributing libraries - distutils

- **Enables programs, modules, and packages to be bundled and unbundled in a standard way**
- Part of the standard library
- **Based on setup.py written by the distributer (see over)**
- **Creating a distribution**
- Compressed file is placed into sub-directory `./dist`

```
C:\product> python setup.py sdist
```

- **Installing a distribution**

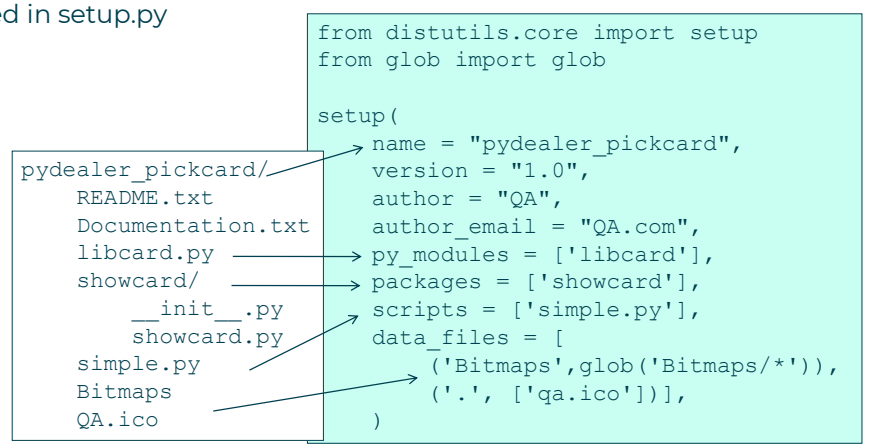
```
C:\product\dist> unzip product-1.0.zip
C:\product\dist> cd product-1.0
C:\product\dist> python setup.py install
```

The **distutils** module is designed to provide a uniform interface for users to create, distribute, and install modules and associated files. From the user's viewpoint, it is based around a script normally called **setup.py**, which is described on the next page. The distribution is usually in the form of a zip file or a gzip tarball, depending on the platform, created in a sub-directory called **`dist`**.
For pure python modules, the **`sdist`** argument for setup.py should be sufficient, **`sdist`** means source distribution. If the distribution includes binary files, such as executables or other platform specific files, then it should be **`bdist`**. A binary distribution will include the .pyc files for the modules.
The argument **`bdist_wininst`** will produce an .exe file which the user has to just double-click on to invoke the Windows installer. This also registers the module, so it can be uninstalled using the control panel. Beware that the generated .exe file can be architecture specific, so that a 64-bit .exe file will not run on a 32-bit Windows installation.

Distributing libraries - distutils

There is a standard way of organising your files

- Described in setup.py



When generating a distribution, the first thing to do is to organise your files in the standard way. If you don't like the standard layout, then you can specify a different one in **`setup.py`**, but that is not worth the effort unless you have a very good reason. The top-level directory does not have to be the name of the distribution, but it would be confusing if it was not.
The next step is to write **`setup.py`**. The example shown does not include all possible combinations, but covers many that are optional. The absolute minimum **`setup.py`** for a single module is:

```
from distutils.core import setup

setup(
    name = "modulename",
    py_modules = ['modulename'],
    )
```

The default version number is 0.0.0, so it is probably best to set a version number as well.