Chapter 25
Python Modules and Packages

25.1 Introduction

Modules and packages are two constructs used in Python to organise larger programs. This chapter introduces modules in Python, how they are accessed, how they are define and how Python finds modules etc. It also explores Python packages and sub-packages.

25.2 Modules

A module allows you to group together related functions, classes and code in general. You can think of a module as being like a library of code (although in fact many libraries are themselves composed of several modules as for example, a library may have optional or extensions to the core functionality).

It is useful to organise your code into modules when the code either becomes large or when you want to reuse some elements of the code base in multiple projects.

Breaking up a large body of code from a single file helps with simplifying code maintenance and comprehensibility of code, testing, reuse and scoping code. These are explored below:

- *Simplicity*—Focussing on a subset of an overall problem helps us to develop solutions that work for the subset and can be combined together to solve the overall problem. This means that individual modules can be simpler than the overall solution.
- *Maintenance*—Modules typically make it easier to define logical boundaries between one body of code and another. This means it is easier to see what comprises a module and to verify that the module works appropriately even

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_25

when modified. It also helps to distinguish one body of code from another so makes it easier to work out where changes should go.

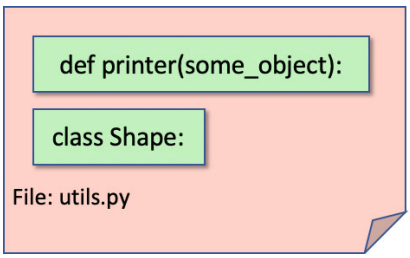
- *Testing*—As one module can be made independent of another module there are less dependencies and cross overs. This means that a module can be tested in isolation and even before other modules, and the overall application, have been written.
- *Reusability*—Defining a function or class on one module means that it is easier to reuse that function or class in another module, as the boundaries between the one module and another are clear.
- *Scoping*—Modules typically also define a namespace, that is a scope within which each function or class is unique. Think of a namespace a bit like a surname; within a classroom there may be several people with the first name 'John', but we can distinguish each person by using their full name, for example 'John Hunt', 'John Jones', 'John Smith' and 'John Brown'; each surname in this example provides a namespace which ensures each John is unique (and can be referenced uniquely).

25.3 Python Modules

In Python a module equates to a file containing Python code. A module can contain

- Functions
- Classes
- Variables
- Executable code
- Attributes associated with the module such as its name.

The name of a module is the name of the file that it is defined in (minus the suffix '.py'). For example, the following diagram illustrates a function and a class defined within a file called `utils.py`:



Thus, the `printer()` function and the class `Shape` are defined in the `utils` module. They can be referenced via the name of the `utils` module.

As an example, let us look at a definition for our `utils` module defined in the file `util.py`:

```
"""This is a test module"""
print('Hello I am the utils module')

def printer(some_object):
    print('printer')
    print(some_object)
    print('done')

class Shape:
    def __init__(self, id):
        self._id = id

    def __str__(self):
        return 'Shape - ' + self._id

    @property
    def id(self):
        """ The docstring for the id property """
        print('In id method')
        return self._id

    @id.setter
    def id(self, value):
        print('In set_age method')
        self._id = id

default_shape = Shape('square')
```

The module has a comment which is at the start of the file—this is useful documentation for anyone working with the module. In some cases, the comment at the start of the module can provide extensive documentation on the module such as what it provides, how to use the features in the module and examples that can be used as a reference.

The module also has some executable code (the `print` statement just after the comment) that will be run when the module is loaded/initialised by Python. This will happen when the module is first referenced in an application.

A variable `default_shape` is also initialised when the module is loaded and can also be referenced outside the module in a similar manner to the module's function and class. Such variables can be useful in setting up defaults or predefined data that can be used by developers working with the module.

25.4 Importing Python Modules

25.4.1 Importing a Module

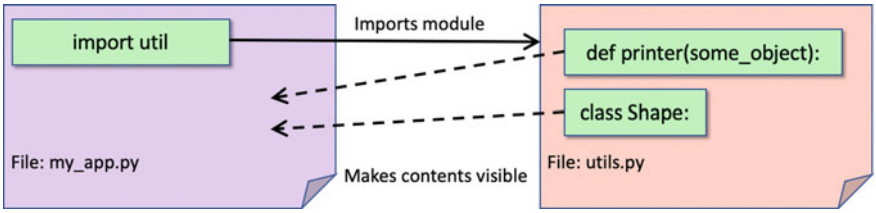
A user defined module is *not* automatically accessible to another file or script; it is necessary to `import` the module. Importing a module makes the functions, classes and variables defined in the module visible to the file they are imported into.

For example, to import all the contents of the `utils` module into a file called `my_app.py` we can use:

```
import utils
```

Note that we do not give the file name (i.e. `utils.py`) instead we give the module name to import (which does not include the `.py`).

The result is as shown below:



Once the definitions within the `utils` module are visible inside `my_app.py` we can use them as if they had been defined in the current file. Note that due to scoping the function, class and variable defined within the `utils` module will be prefixed by the name of the module; i.e. `utils.printer` and `utils.Shape` etc.:

```
import utils

utils.printer(utils.default_shape)
shape = utils.Shape('circle')
utils.printer(shape)
```

When we run the `my_app.py` file we will produce the following output:

```
Hello I am the utils module
printer
Shape - square
done
printer
Shape - circle
```

Notice that the first line of the output is the output from the `print` statement in the `utils` module (i.e. the line `print('Hello I am the utils module')`) this illustrates the ability in Python to define behaviour that will be run when a module is loaded; typically this might execute some housekeeping code or set up behaviour required by the module. It should be noted that the module is only initialised the first time that it is loaded and thus the executable statements are only run once.

If we forgot to include the `import` statement at the start of the file, then Python would not know to use the `utils` module. We would thus generate an error indicating that the `utils` element is not known: `NameError: name 'utils' is not defined`.

There can be any number of modules imported into a file; these can be both user defined modules and system provided modules. The modules can be imported via separate import statements or by supplying a comma separated list of modules:

```
import utils
import support
import module1, module2, module3
```

A common convention is to place all import statements at the start of a file; however, this is only a convention and `import` statements can be placed anywhere in a file prior to where the imported module's features are required. It is a common enough convention however, that tools such as PyCharm will indicate a style issue if you do not put the imports at the start of the file.

25.4.2 Importing from a Module

One issue with the previous example is that we have had to keep referring to the facilities provided by the `utils` module as `utils.<thing of interest>`; which while making it very clear that these features come from the `utils` module, is a little tedious. It is the equivalent of referring to a person via their full name every time we speak to them. However, if they are the only other person in the room then we don't usually need to be so formal; we could just use their first name.

A variant of the `import` statement allows us to import *everything* from a particular module and remove the need to prefix the modules functions or classes with the module name, for example:

```
from <module name> import *
```

Which can be read as *from <module name> import everything* in that module and make it directly available.

```
from utils import *
printer(default_shape)
shape = Shape('circle')
printer(shape)
```

As this shows we can reference `default_shape`, the `printer()` function and the `Shape` class directly. Note the `'*'` here is often referred to as a wildcard; meaning that it represents everything in the module.

The problem with this form of `import` is that it can result in name clashes as it brings into scope all the elements defined in the `utils` module. However, we may only actually be interested in the `Shape` class; in which case we can choose to only bring that feature in, for example:

```
from utils import Shape
s = Shape('rectangle')
print(s)
```

Now only the `Shape` class has been imported into the file and made directly available.

You can even give an alias for an element being imported from a module using the `import` statement. For example, you can alias the whole package:

```
import <module_name> as <alternative_module_name>
```

for example:

```
import utils as utilities
utilities.printer(utilities.default_shape)
```

In this example, instead of referring to the module we are importing as `utils`, we have given it an alias and called it `utilities`.

We can also alias individual elements of a module, for example a function can be given an alias, as can a variable or a class. The syntax for this is

```
from <module_name> import <element> as <alias>
```

For example:

```
from utils import printer as myfunc
```

In this case the `printer` function in the `utils` module is being aliased to `myfunc` and will thus be known as `myfunc` in the current file.

We can even combine multiple `from` imports together with some of the elements being imported having aliases:

```
from utils import Shape, printer as myfunc
s = Shape('oval')
myfunc(s)
```

25.4.3 Hiding Some Elements of a Module

By default, any element in a module whose name starts with an underbar (`'_'`) is hidden when a wildcard import of the contents of a module is performed.

In this way certain named items can be hidden unless they are explicitly imported. Thus, if our `utils` module now included a function:

```
"""This is a test module"""
print('Hello I am the utils module')

# as before

def _special_function():
    print('Special function')
```

And then we tried to import the whole module using the wildcard import and access `_special_function`:

```
from utils import *
_special_function()
```

We will get an error:

```
NameError: name '_special_function' is not defined
```

However, if we explicitly import the function then we can still reference it:

```
from utils import _special_function
_special_function()
```

Now the code works:

```
Hello I am the utils module
Special function
```

This can be used to hide features that are either not intended to be used externally from a module (developers then use them at their own peril) or to make advanced features only available to those who really want them.

25.4.4 Importing Within a Function

In some cases, it may be useful to limit the scope of an `import` to a function; thus, avoiding any unnecessary use of, or name clashes with, local features.

To do this you merely add an `import` into the body of a function, for example:

```
def my_func():
    from util import Shape
    s = Shape('line')
```

In this case the `Shape` class is only accessible within the body of `my_func()`.

25.5 Module Properties

Every module has a set of properties that can be used to find what features it provides, what its name is, what (if any) its documentation string is etc.

These properties are considered special as they all start, and end, with a double underbar (`'__'`). These are:

- `__name__` the name of the module
- `__doc__` the doctoring for the module
- `__file__` the file in which the module was defined.

You can also obtain a list of the contents of a module once it has been imported using the `dir(<module-name>)` function. For example:

```
import utils
print(utils.__name__)
print(utils.__doc__)
print(utils.__file__)
print(dir(utils))
```

Which produces:

```
Hello I am the utils module
utils
This is a test module
utils.py
['Shape', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__',
'_special_function', 'default_shape', 'printer']
```

Note that the executable `print` statement is still executed as this is run when the module is loaded into Python's current runtime; even though all we do it then access some of the module's properties.

Mostly module properties are used by tools to help developers; but they can be a useful reference when you are first encountering a new module.

25.6 Standard Modules

Python comes with many built-in modules as well as many more available from third parties.

Of particular use is the `sys` module, which contains a number of data items and functions that relate to the execution platform on which a program is running.

Some of the `sys` module's features are shown below, including `sys.path()`, which lists the directories that are searched to resolve a module when an `import` statement is used. This is a writable value, allowing a program to add directories (and hence modules) before attempting an import.

```
import sys
print('sys.version: ', sys.version)
print('sys.maxsize: ', sys.maxsize)
print('sys.path: ', sys.path)
print('sys.platform: ', sys.platform)
```

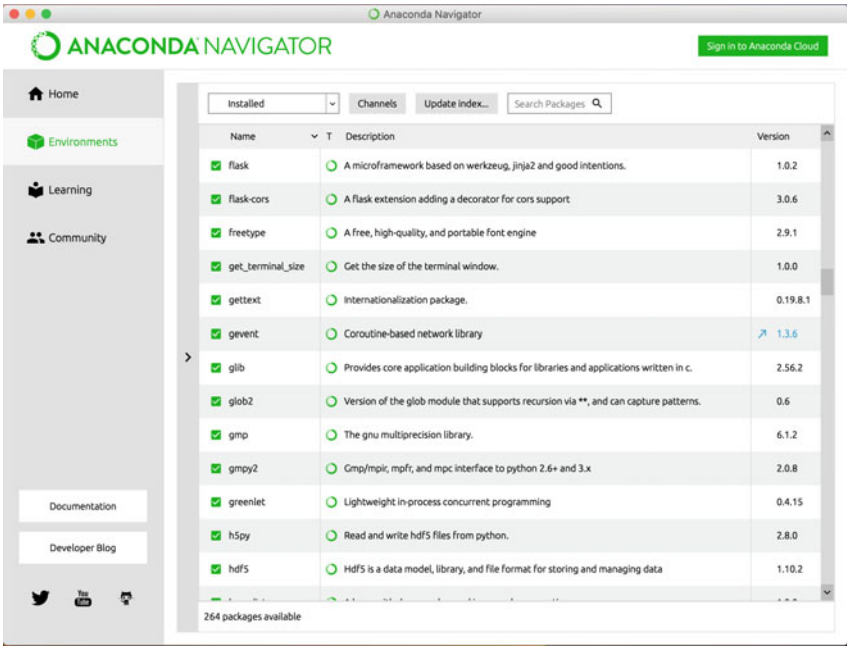
Which produces the following output on a Mac:

```
[Clang 6.0 (clang-600.0.57)]
sys.maxsize:  9223372036854775807
sys.path:  ['/pythonintro/modules', '/workspaces/pycharm', '/
Library/Frameworks/Python.framework/Versions/3.7/lib/
python37.zip', '/Library/Frameworks/Python.framework/Versions/
3.7/lib/python3.7', '/Library/Frameworks/Python.framework/
Versions/3.7/lib/python3.7/lib-dynload', '/Library/Frameworks/
Python.framework/Versions/3.7/lib/python3.7/site-packages']
sys.platform:  Darwin
```

A common module management tool is known as *Anaconda*. Anaconda (which is widely used particularly within the Data Science and Data Analytics field) is shipped with a large number of common third party Python libraries/modules. Using Anaconda avoids the need to download separate modules; instead Anaconda acts as a repository of (most) modules and means that accessing these modules is as simple as importing them into your code.

You can download Anaconda from https://www.anaconda.com/download.

To see the available Python libraries in Anaconda use the Anaconda Navigator:



25.7 Python Module Search Path

One point that we have glossed over so far is how does Python find these modules? The answer is that it uses a special environment variable called the PYTHONPATH. This is a variable that can be set up prior to running Python that tells it where to look for to find any named modules.

It is hinted at in the previous section where the sys module's path variable is printed out. On the machine that this code was run on the output of this variable was:

```
sys.path:  ['/pycharm/pythonintro/modules', '/workspaces/
pycharm', '/Library/Frameworks/Python.framework/Versions/3.7/
lib/python37.zip', '/Library/Frameworks/Python.framework/
Versions/3.7/lib/python3.7', '/Library/Frameworks/
Python.framework/Versions/3.7/lib/python3.7/lib-dynload', '/
Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/
site-packages']
```

This is actually a list of the locations that Python would look into find a module; these include the PyCharm project holding the modules related to the examples used in this book (which is the current directory), then a top level PyCharm location and a series of Python locations (where locations equal directories on the host machine).

Python will search though each of these locations in turn to find the named module; it will use the first module it finds. The actual search algorithm is:

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On Unix/Linux this default path is normally /usr/local/lib/python/.

One point to note about this search order is that it is possible to hide a system provided module by creating a user defined one and giving it the same name as the system module; this is because your own module will be found first and will hide the system provided one.

In this case the PYTHONPATH will be used to find built in Python modules under the Python installation, while our own user defined modules will be found within the PyCharm project workspace directories.

It is also possible to override the default PYTHONPATH variable. It is what is known as an environment variable and so it can be set as a Unix/Linux or Windows operating system environment variable which can then be picked up by your Python environment. The syntax used to set the PYTHONPATH depends on whether you are using Windows or Unix/Linux as it is set at the operating system level:

Here is a typical PYTHONPATH from a Windows system:

```
set PYTHONPATH = c:\python30\lib;
```

And here is a typical PYTHONPATH from a Unix/Linux system:

```
set PYTHONPATH = /usr/local/lib/python
```

25.8 Modules as Scripts

Any Python file is not only a module but also a Python script or program. This means that it can be executed directly if required. For example, the following is the contents of a file called module1.py:

```
"""This is a test module"""
print('Hello I am module 1')

def f1():
    print('f1[1]')

def f2():
    print('f2[1]')

x = 1 + 2
print('x is', x)
f1()
f2()
```

When this file is run directly, or loaded into the Python REPL, then the free-standing code will be executed and the output generated will be:

```
Hello I am module 1
x is 3
f1[1]
f2[1]
```

This looks fine until you try to use module1 with your own code; the free-standing code will still execute this if we now write:

```
import module1
module1.f1()
```

Where you might expect only to see the result of running the `f1()` function from `module1` you actually get:

```
Hello I am module 1
x is 3
f1[1]
f2[1]
f1[1]
```

The first 4 lines are run when `module1` is loaded as they are free standing executable statements within the module.

We can of course remove the free-standing code; but what if we sometimes want to run `module1` as a script/program and sometimes use it as a module imported into other modules?

In Python we can distinguish between when a file is loaded as a module and when it is being run as a standalone script/program. This is because Python sets the module property `__name__` to the name of the module when it is being loaded as a module; but if a file is being run as a standalone script (or the entry point of an application) then the `__name__` is set to the string `__main__`. This is partly because historically `main()` has been the entry point for applications in numerous other languages such as C, C++, Java and C#.

We can now determine whether a module is being loaded or run as a script/main application by checking the `__name__` property of the module (which is directly accessible from within a module). If the module is the `__main__` entry point then run some code; if it is not then do something else.

For example,

```
"""This is a test module"""
print('Hello I am module 1')

def f1():
    print('f1[1]')

def f2():
    print('f2[1]')

if __name__ == '__main__':
    x = 1 + 2
    print('x is', x)
    f1()
    f2()
```

Now the code within the `if` statement will only be executed when this module is loaded as the starting point for an application/script. Note that the `print` statement at the top of the module will still be executed in both scenarios; this can be useful as it allows set up or initialisation behaviour to still be executed where appropriate.

A common pattern, or idiom is to place the code to be run when a file is being loaded directly (rather than as a module) in a function called `main()` and to call that function from within the `if` statement. This helps to clarify which behaviour is intended to run when, thus our module's final version is:

```
"""This is a test module"""
print('Hello I am module 1')

def f1():
    print('f1[1]')

def f2():
    print('f2[1]')

def main():
    x = 1 + 2
    print('x is', x)
    f1()
    f2()

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

This version is what would now be called *idiomatic* Python or *Pythonic* in style.

25.9 Python Packages

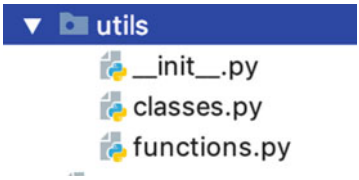
25.9.1 Package Organisation

Python allows developers to organize modules together into packages, in a hierarchical structure based on directories.

A package is defined as

- a *directory* containing one or more Python source files and
- an *optional* source file named `__init__.py`. This file may also contain code that is executed when a module is imported from the package.

For example, the following picture illustrates a package `utils` containing two modules `classes` and `functions`.



In this case the `__init__.py` file contains package level initialisation code:

```
print('utils package')
```

The contents of the `__init__.py` file will be run once, the first time either module within the package is referenced.

The `functions` module then contains several function definitions; while the `classes` module contains several class definitions.

We refer to elements of the package relative to the package name—as shown below:

```
from utils.functions import *
f1()
from utils.classes import *
p = Processor()
```

Here we are importing both the `functions` module and the `classes` module from the `utils` package. The function `f1()` is defined in the `functions` module while the `Processor` class is defined within the `classes` module.

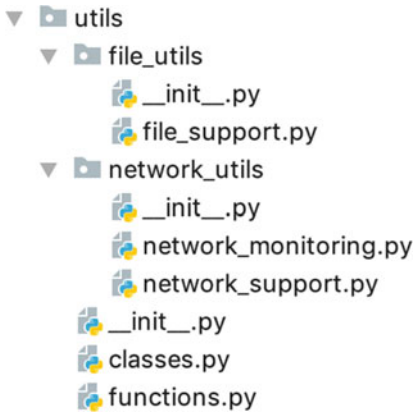
You can use all the *from* and *import* styles we have already seen, for example you can import a function from a module in a package and give it an alias:

```
from util.functions import f1 as myfunc
myfunc()
```

It is possible to import all the modules from a package by simply importing the package name. If you wish to provide some control over what is imported from a package when this happens you can define a variable `all` in the `__init__.py` file that will indicate what will be imported in this situation.

25.9.2 Sub Packages

Packages can contain sub packages to any depth you require. For example, the following diagram illustrates this idea:



In the above diagram `utils` is the root package, this contains two sub packages `file_utils` and `network_utils`. The `file_utils` package has an initialisation file and a `file_support` module. The `network_utils` package also has a package initialisation file and two modules; `network_monitoring` and `network_support`. Of course, the root package also has its own initialisation file and its own modules `classes` and `functions`.

To import the sub package, we do the same as before but each package in the *path* to the module is separate by a dot, for example:

```
import utils.file_utils.file_support
```

or

```
from utils.file_utils.file_support import file_logger
```

25.10 Online Resources

See the Python Standard Library documentation for:

- https://docs.python.org/3/tutorial/modules.html#standard-modules the standard modules.
- https://docs.python.org/3/tutorial/modules.html#packages packages.

- https://docs.python.org/3/tutorial/stdlib.html brief tour of the standard library part 1.
- https://docs.python.org/3/tutorial/stdlib2.html brief tour of the standard library part 2.
- https://pymotw.com/3/ the Python Module of the Week site listing very many modules and an extremely useful reference.

25.11 Exercise

The aim of this exercise is to create a module for the classes you have been developing.

You should move your `Account`, `CurrentAccount`, `DepositAccount` and `BalanceError` classes into a separate module (file) called `accounts`. Save this file into a new Python package called `fintech`.

Separate out the test application from this module so that you can import the classes from the package.

Your test application will now look like:

```
import fintech.accounts as accounts

acc1 = accounts.CurrentAccount('123', 'John', 10.05, 100.0)
acc2 = accounts.DepositAccount('345', 'John', 23.55, 0.5)
acc3 = accounts.InvestmentAccount('567', 'Phoebe', 12.45, 'high
risk')

print(acc1)
print(acc2)
print(acc3)

acc1.deposit(23.45)
acc1.withdraw(12.33)
print('balance:', acc1.balance)

print('Number of Account instances created:',
accounts.Account.instance_count)

try:
    print('balance:', acc1.balance)
    acc1.withdraw(300.00)
    print('balance:', acc1.balance)
except accounts.BalanceError as e:
    print('Handling Exception')
    print(e)
```

You could of course also use `from accounts import *` to avoid prefixing the accounts related classes with accounts.