Modules and Imports

modules and imports

- Import syntax variations
- Namespaces and variable lookups
- sys.modules and the import cache
- Module level functionality: `__dir__` and `__getattr__`
- Packages and the filesystem
- Relative import syntax
- Module reloading (how to do it and why not to do it)
- Circular imports, avoiding and fixing
- Executable modules and packages
- `__import__` and `importlib`

Modules

- Every Python source file is a module

```
# foo.py
def grok(a):
    ...
def spam(b):
    ...
```

- The import statement loads and executes a module

```
import foo

a = foo.grok(2)
b = foo.spam('Hello')
...
```

Module Objects

- Modules are objects

```
>>> import foo
>>> foo
<module 'foo' from 'foo.py'>
```

- A namespace for the definitions inside

```
>>> foo.grok(2)
```

- Actually a layer on top of dictionaries (module globals)

```
>>> foo.__dict__['grok']
<function grok at 0x7f733a914670>
```

Special Variables

- A few special variables defined automagically in a module

```
__file__        # Name of the source file
__name__        # Name of the module
__doc__         # Module docstring
```

- We can use __name__ to tell if we have been imported or are running as the main module (the main script or program)

Import Implementation

- Import in a nutshell (pseudocode)

```python
import types

def import_module(name):
    # locate the module and get source code
    filename = find_module(name)
    code = open(filename).read()
    # Create the enclosing module object
    mod = types.ModuleType(name)
    # Run it
    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

- Import finds and executes the module source file to create the module object

import Statement

- import executes the <u>entire</u> module

```python
# bar.py
import foo
```

- After the import the module name is a reference to the module object inside the module that did the import

```python
# bar.py
import foo

foo.grok(2)
```

bar.__dict__

```
{
    ...
    "foo": <module 'foo'>,
    ...
}
```

Module Cache

- Each module is loaded only <u>once</u>
- Repeated imports return a reference to the same module object
- sys.modules is a dict of all loaded modules

```
>> import sys
>>> list(sys.modules)
['sys', 'builtins', '_frozen_importlib', '_imp',
'_warnings', '_io', 'marshal', 'posix',
'_frozen_importlib_external', '_thread', ... ]
```

Import Caching

- Import pseudocode with module caching

```python
import types
import sys

def import_module(name):
    # Check for cached module
    if name in sys.modules:
        return sys.modules[name]
    filename = find_module(name)
    code = open(filename).read()
    mod = types.ModuleType(name)
    sys.modules[name] = mod
    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

- There are many more details but this is most of the work

- See the `importlib` module for more

from module import

- Import selected symbols with "from module import ..."

```
# bar.py
from foo import grok

grok(2)
```

- Useful for frequently used names
- Slightly faster as it avoids the attribute lookup of "foo.grok"

Note: This doesn't change how import works, the entire source file is still executed and the module cached but you only have references to the specific objects imported

from module import *

- Imports *all* the symbols from a module

```
# bar.py
from foo import *

grok(2)
spam('Hello')
...
```

- "import *" *doesn't* import private names from a module, those that start with an underscore (_)
- Not recommended! Namespace pollution!
- Useful for the interactive interpreter and for namespace packages (__init__.py)

Module Reloading

- Modules can sometimes be reloaded

```
>>> import foo
...
>>> import importlib
>>> importlib.reload(foo)
<module 'foo' from 'foo.py'>
```

- Reloading re-executes the source code over the top of the existing module dictionary

```
# pseudocode
def reload(mod):
    code = open(mod.__file__, 'r').read()
    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

Module Reloading Dangers

- Problem: Existing instances of classes will still have a reference to the old class object
- Problem: Anything imported with "from module import name" will still use the old definition
- Problem: Anything that uses super or does type checks will fail
- Reloading is not for production code but very useful for development where modules change rapidly

Locating Modules

- When looking for modules Python first looks in the same directory as the source file that's executing the import
- If the module isn't found there sys.path contains a list of places that Python checks (in order)

```
>>> import sys
>>> sys.path
['', '/usr/lib/python310.zip', '/usr/lib/python3.10',
'/usr/lib/python3.10/lib-dynload',
'/home/michael/.local/lib/python3.10/site-packages',
'/usr/local/lib/python3.10/dist-packages',
'/usr/lib/python3/dist-packages']
```

Module Search Path

- sys.path contains the search paths for import
- You can manually adjust it if you need to

```
import sys
sys.path.append('/project/foo/pyfiles')
```

- Paths can also be added via environment variables

```
$ export PYTHONPATH=/home/user/code
```

- The directory added will now be at the start of sys.path when you run Python

Organising Libraries

- It is standard practise for Python libraries to be organised as a hierarchical set of modules that sit under a top level package name

```
packagename
packagename.foo
packagename.bar
packagename.utils
packagename.utils.spam
packagename.utils.grok
packagename.parsers
packagename.parsers.xml
packagename.parsers.json
...
```

Creating a Package

- To create the module library hierarchy, organize files on the filesystem in a directory with the desired structure

```
packagename/
            foo.py
            bar.py
            utils/
                  spam.py
                  grok.py
            parsers/
                    xml.py
                    json.py
            ...
```

Creating a Package

- Optionally add __init__.py files to each directory

```
packagename/
            __init__.py
            foo.py
            bar.py
            utils/
                  __init__.py
                  spam.py
                  grok.py
            parsers/
                    __init__.py
                    xml.py
                    json.py
            ...
```

Using a Package

- You can now import from your package

```
import packagename.foo
import packagename.parsers.xml
from packagename.parsers import xml
```

- Almost everything should work as it did before but your import statements can have multiple levels

Fixing Relative Imports

- Relative imports between submodules don't work

```
spam/
    __init__.py
    foo.py
    bar.py
```

```
# bar.py
import foo    # import fails
```

- The issue: resolving name clashes between submodules and top level packages

```
spam/
    __init__.py
    os.py
    bar.py
```

```
# bar.py
import os    # ??? stdlib?
```

- imports are always "absolute" (from the top level)

Absolute Imports

- One approach: always use absolute imports

```
spam/
     __init__.py
     foo.py
     bar.py
```

- Example:

```
# bar.py
from spam import foo
```

- Notice the use of the top level package name

Package Relative Imports

- Consider a package

```
spam/
    __init__.py
    foo.py
    bar.py
    grok/
        __init__.py
        blah.py
```

- Package relative imports

```python
# bar.py
from . import foo              # Imports ./foo.py
from .foo import name          # Load a specific name


from .grok import blah         # Imports ./grok/blah.py
```

Package Environment

- Packages define a few useful variables

```
__package__      # Name of the enclosing package
__path__         # Search path for subcomponents
```

- Example:

```
>>> import xml
>>> xml.__package__
'xml'
>>> xml.__path__
['/usr/local/lib/python3.10/xml']
```

- Useful if code needs to know about its enclosing environment

__init__.py Usage

- What are you supposed to do in these files?
- __init__.py provides the top level namespace for the package/sub-package
- Main use: stitching together multiple files (the submodules) into a "unified" top-level import

Tip: Don't put lots of code in __init__.py, use it to create the namespace, use helpfully named modules for the actual code.

Module Assembly

- Consider two submodules in a package

```
spam/
    foo.py
```

➡

```python
# foo.py

class Foo:
    ...
    ...
```

```
    bar.py
```

➡

```python
# bar.py

class Bar:
    ...
    ...
```

- Suppose we wanted to combine them for import

Module Assembly

- Combine in __init__.py

```
spam/
    foo.py
```

```python
# foo.py

class Foo:
    ...
...
```

```
    bar.py
```

```python
# bar.py

class Bar:
    ...
...
```

```
    __init__.py
```

```python
# __init__.py

from .foo import Foo
from .bar import Bar
```

Module Assembly

- Users see a single unified top-level package

```python
import spam

foo = spam.Foo()
bar = spam.Bar()
...
```

- The internal split across submodules, the implementation details, are hidden from the user

Case Study

- The "asyncio" module
- It's actually a package with many sub-components
- If we look in the __init__.py we can see all the submodules that provide the implementation

```
# This relies on each of the submodules having an __all__
variable.
from .base_events import *
from .coroutines import *
from .events import *
from .exceptions import *
from .futures import *
from .locks import *
from .protocols import *
from .runners import *
from .queues import *
...
```

Controlling Exports

- Submodules should define __all__

```python
# foo.py

__all__ = ['Foo']

class Foo:
    ...
```

```python
# bar.py

__all__ = ['Bar']

class Bar:
    ...
```

- This controls 'from module import *'
- Which makes it easier to create __init__.py

```python
# __init__.py
from .foo import *
from .bar import *

__all__ = [ *foo.__all__, *bar.__all__ ]
```

Module Splitting

- Suppose you have a large module

```python
# spam.py

class Foo:
    ...
    ...

class Bar:
    ...
    ...
```

- You want to refactor this into multiple files for sanity
- But you want to keep the external API the same as the single file for backwards compatibility

Module Splitting

- Step 1: turn it into a directory with multiple files

```
spam/
    foo.py
```

```
# foo.py

class Foo:
    ...
    ...
```

```
    bar.py
```

```
# bar.py

class Bar:
    ...
    ...
```

- Split the code across the submodules as you wish
- Fix any dependencies between submodules with appropriate imports (not relative imports!)

Module Splitting

- Step 2: stitch back together in __init__.py

```
spam/
    foo.py
```

```python
# foo.py

class Foo:
    ...
    ...
```

```
    bar.py
```

```python
# bar.py

class Bar:
    ...
    ...
```

```
    __init__.py
```

```python
# __init__.py

from .foo import Foo
from .bar import Bar
```

Circular Imports

- Circular imports are a common problem within submodules, modules that depend on each other

```python
# spam/base.py

from . import child

class Base:
    ...
```

```python
# spam/child.py

from .base import Base

class Child(Base):
    ...
```

- Follow the control-flow
- Definition order matters!

Circular Imports

- Moving the import can fix the problem

```python
# spam/base.py

class Base:
    ...

from . import child
```

```python
# spam/child.py

from .base import Base

class Child(Base):
    ...
```

- Alternative fixes:
  - An "inner import" inside a function/method
  - Refactoring the shared code into a third module
- Better yet, avoid circular imports!

Main Modules

- python -m *module*
- Runs the specified module as a script

```
spam/
     __init__.py
     foo.py
     bar.py

$ python -m spam.foo      # runs spam/foo.py
```

- Can be used to ship tools and applications as part of a package
- Many standard library modules can be run this way (e.g. unittest, gzip, idlelib, timeit, asyncio, antigravity, etc)

Main Entry Point

- __main__.py designates an entry point
- Makes the package itself executable

```
spam/
    __init__.py
    __main__.py            # starting module
    foo.py
    bar.py

$ python -m spam           # runs spam/__main__.py
```

- Can also be done with subpackages, ship multiple tools within a package!