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

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

- The import statement loads and executes a module

```python
import foo

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

- Even the main script/program is run as a module!

Namespaces

- A module is a collection of named values (i.e. it's said to be a "namespace")
- The names are all the global variables and functions defined in the source file
- After import the module name can be used as a prefix

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

- The module name is tied to the source file name (foo → foo.py)

Global Definitions

- Everything defined in the "global" scope is what populates the module namespace

```python
# foo.py
x = 42
def grok(a):
    ...
```

```python
# bar.py
x = 37
def spam(a):
    ...
```

These definitions of x are different

- Different modules can use the same names and those names don't conflict with each other, because each module forms a separate namespace

Modules as Environments

- Modules form an enclosing environment for <u>all</u> of the code defined inside

```python
# foo.py
x = 42

def grok(a):
    print(x)
```

global variables are always bound to the enclosing module (the same file)

- Each source file is it's own little universe
- This is great!
- What happens in a module stays in a module

Module Objects

- Modules are <u>objects</u>

```
>>> 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)

Main Functions

- In many programming languages there is a concept of a "main" function or method

```
/* C/C++ */
int main(int argc, char *argv[]) {
    ...
}

/* Java */
class myprog {
public static void main(String args[]) {
        ...
    }
}
```

- It's the code that runs when the program is launched
- The entrypoint

Main Module

- Python has no "main" function or method
- Instead, there's a "main" module
- It's the source file that is run first:

```
$ python prog.py
...
```

- Whatever program or script is given at startup is run as the "main" module

__main__ check

- Every module has a name, the `__name__` variable
- The main module is called `"__main__"`
- It's standard practise for modules that *can* run as a main script to use this convention for the entrypoint

```
# prog.py
...
if __name__ == '__main__':
    # Running as the main program
    ...
    statements
    ...
```

- The code inside the __name__ check is the entrypoint for the program

__main__ check

- Important: Any file can run as main *or* be imported
- Consider this simple code:

```
# foo.py
print(__name__)
```

- It behaves differently when run and when imported:

```
$ python foo.py
__main__

>>> import foo
foo
```

- As a general rule you don't want scripting tasks to run when you import code, thus the __main__ check:

```
if __name__ == '__main__':
    # Does not execute if loaded with import
    ...
```

Import Implementation

- Import in a nutshell (pseudocode)

```
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 entire 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 ..."

```python
# 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

```python
# 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)

Dynamic Objects

- Objects may implement dynamic methods to control what attributes are available
- `__getattr__` and `__getattribute__` for attribute lookup
  - `__getattr__` is only called for attributes that don't exist
  - `__getatttribute__` for every lookup (implemented on object by default)
- `__delattr__` for attribute deletion
- `__setattr__` for attribute setting
- `__dir__` returns a list of all available attributes and is called by dir
- Used rarely, for things like mock or proxy objects
- Modules may implement these as functions!

Proxy Object

- Using a closure to proxy attribute access

```python
def proxy(thing: Any, attrs: list | set):
    attrs = set(attrs)
    class ProxyThing:
        def __getattr__(self, name):
            if name in attrs:
                return getattr(thing, name)
            raise AttributeError(name)
        def __setattr__(self, name, value):
            if name in attrs:
                return setattr(thing, name, value)
            raise AttributeError(name)
        def __delattr__(self, name, value):
            if name in attrs:
                return setattr(thing, name, value)
            raise AttributeError(name)
        def __dir__(self):
            return list(attrs)
    return ProxyThing()
```

Proxy Objects

- Proxied objects only have the attributes we specify

```
>>> h = proxy(open('foo.py'), ['read', 'close'])
>>> dir(h)
['close', 'read']
>>> h.read()
'print(__name__)'
>>> h.close()
>>> h.seek(0)
Traceback (most recent call last):
  ...
    raise AttributeError(name)
AttributeError: seek
```

But beware:

```
>>> import inspect
>>> inspect.getclosurevars(h.__getattr__)
ClosureVars(nonlocals={'attrs': {'close', 'read'}, 'thing':
<_io.TextIOWrapper name='foo.py' mode='r' encoding='UTF-8'>}
...)
```

Dynamic Modules

- Lazy loading with dynamic functions

```python
_attrs = ['csv', 'importlib', 'inspect']

def __dir__():
    return _attrs

def __getattr__(name):
    if name not in _attrs:
        raise AttributeError(name)

    print(f'Importing {name!r} for the first time')
    import sys
    module = sys.modules[__name__]

    attr = __import__(name)
    setattr(module, name, attr)
    return attr
```

Lazy Loading with a Dynamic Module

- `dir(module)` calls the `module.__dir__` function
- `module.attr` calls the `module.__getattr__` function

```
>>> import dynamic
>>> dir(dynamic)
['csv', 'importlib', 'inspect']
>>> dynamic.csv
Importing 'csv' for the first time
<module 'csv' from '/usr/lib/python3.10/csv.py'>
>>> dynamic.csv
<module 'csv' from '/usr/lib/python3.10/csv.py'>
```

- Looking up a module attribute triggers the lazy loading, and then sets the attribute on the module so `__getattr__` won't be called next lookup

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

```python
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

```
# 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
    bar.py
    __init__.py
```

```python
# foo.py

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

```python
# bar.py

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

```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

(*example with stock.py*)

Module Splitting

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

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

```python
# foo.py

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

```python
# 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!