Writing Python Libraries

Import Statements and Packaging

Basics

A Python file is called either a **script** or a **module**, depending on how it's run:

- **Script:** Run file as a top-level script
  - `python file.py`
  - `__name__ == "__main__"`

Field containing module name

- **Module:** Import file as a module
  - `python -m package.file`
  - `import package.file` (inside some other file)
  - `__name__ == "package.file"`

Run a file as a module

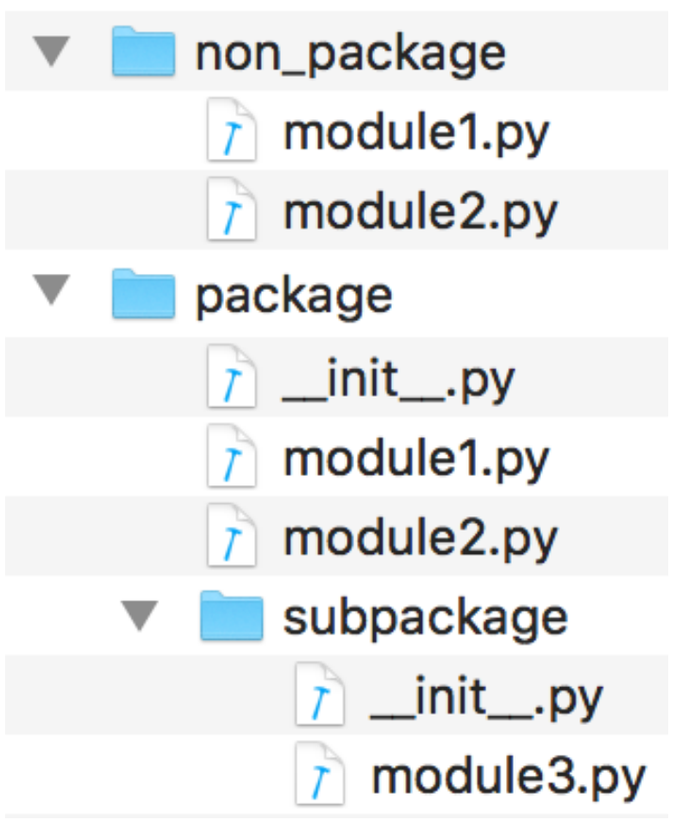
Name depends on root package

Python **packages** are collections of modules. In a directory structure, in order for a folder containing Python files to be recognized as a package, an `__init__.py` file is needed (even if empty).

If a module's name has no dots, it is not considered to be part of a package.

Package Basics

Python **packages** are collections of modules. In a directory structure, in order for a folder containing Python files to be recognized as a package, an `__init__.py` file is needed (even if empty).



Cannot be accessed from root directory using `non_package.module1`

Can be accessed from root directory using `package.subpackage.module3`

Installable Packages

Then the package can be installed by running:

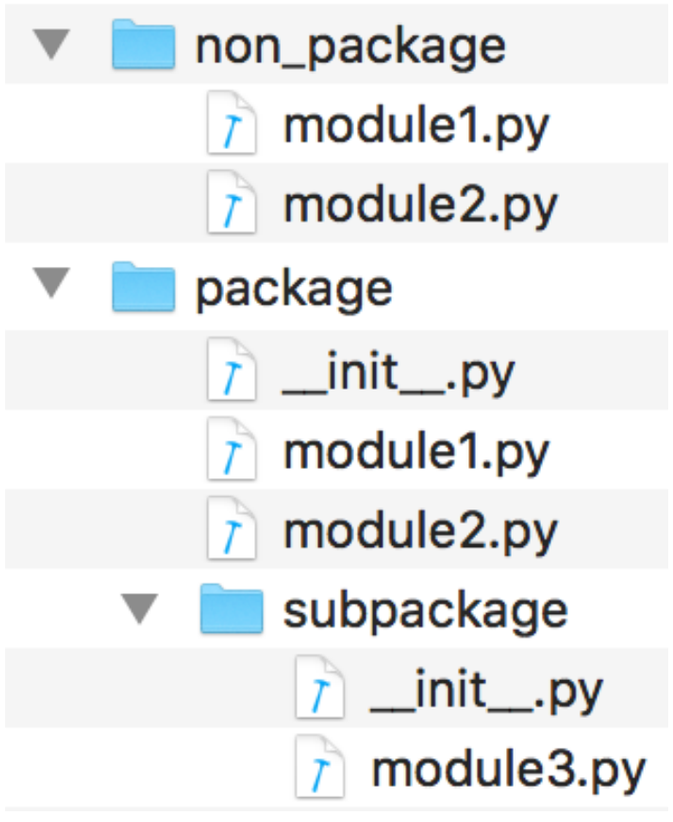
- **`python setup.py install`**
  - This command will install the package in the site-packages directory of the current Python distribution so it can be imported in any Python file using simply: `import project`

- **`python setup.py develop`**
  - This command will install symbolic links to the current package source code in the site-packages directory of the current Python distribution so it can be imported in any Python file using simply: `import project`
  - *Any changes made to the local project files, will be reflected in the installed version of the project*

The `--user` option can optionally be used to install in the current user site-packages directory instead of the system site-packages directory.

Import Basics

Packages and **modules** can be imported in other Python files. Absolute imports are relative to every path in the module search path (`sys.path`) for the packages along with the current directory.



`module2` shall use:
`import module1`

`module1` shall use:
`import subpackage.module3`

Relative Imports

- Relative imports use the module's *name* to determine where it is in a package.

  If `__name__ == “package.subpackage.module”`, then:

  `from .. import other`, resolves to a module with `__name__ == “package.other”`

- `__name__` must have at least as many dots as there are in the import statement.
- If `__name__` has no dots (“`__main__`”), then a `“relative-import in non-package”` error is raised.

If you use relative imports in a Python file and you want to run it use the command: `python -m package.subpackage.module`

Package Name Space

When a Python package is imported, we want to be able to define its name space. This is the set of names (modules, packages, functions, fields, or classes) that this package contains.

Sometimes we might want to **expose names** of a sub-package to the root package, for convenience. For example: `numpy.core.ndarray -> numpy.ndarray`

We can do that using:

- `__all__` field of modules
- `__init__.py` file of packages

Care must always be taken to prevent name space pollution and collisions (i.e., overloaded names).

__all__ Field

The `__all__.py` field can be used to specify which symbols of a module to export. The exported symbols are the ones imported when * is used.

If omitted, all names *not starting* with an underscore (_) are exported.

module.py

```
1   __all__ = ['fn1', 'fn2']
2
3   def fn1(*args, **kwargs):
4     pass
5
6   def fn2(*args, **kwargs):
7     pass
8
9   def fn3(*args, **kwargs):
10    pass
11
12  def _fn4(*args, **kwargs):
13    pass
```

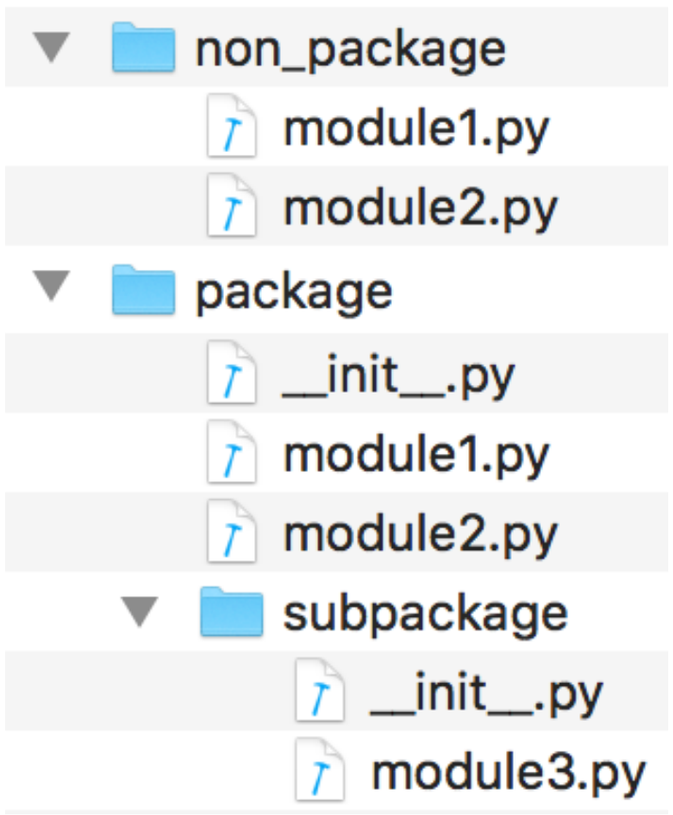
```
1   from module import *
```

Imports `fn1`, `fn2`, `fn3`
if `__all__` is omitted

Imports `fn1`, `fn2`
if `__all__` is specified

__init__.py

The **`__init__.py`** file can be used to export module or sub-package symbols to the package namespace.



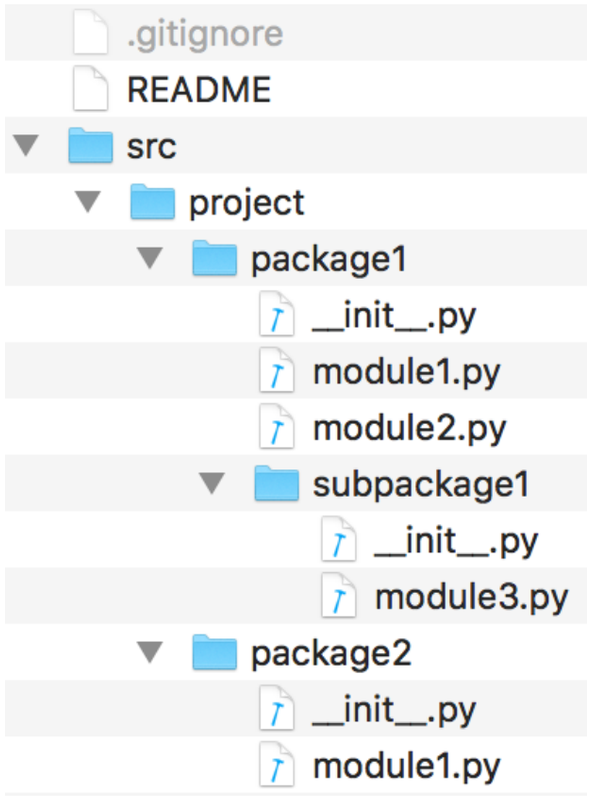
Common Pattern

```
1  from . import subpackage
2
3  from subpackage import *
4
5  __all__ = ['subpackage']
6  __all__.extend(subpackage.__all__)
```

Exposes `module3` to the `package` name space

Common Practices

- Python project directory structure:



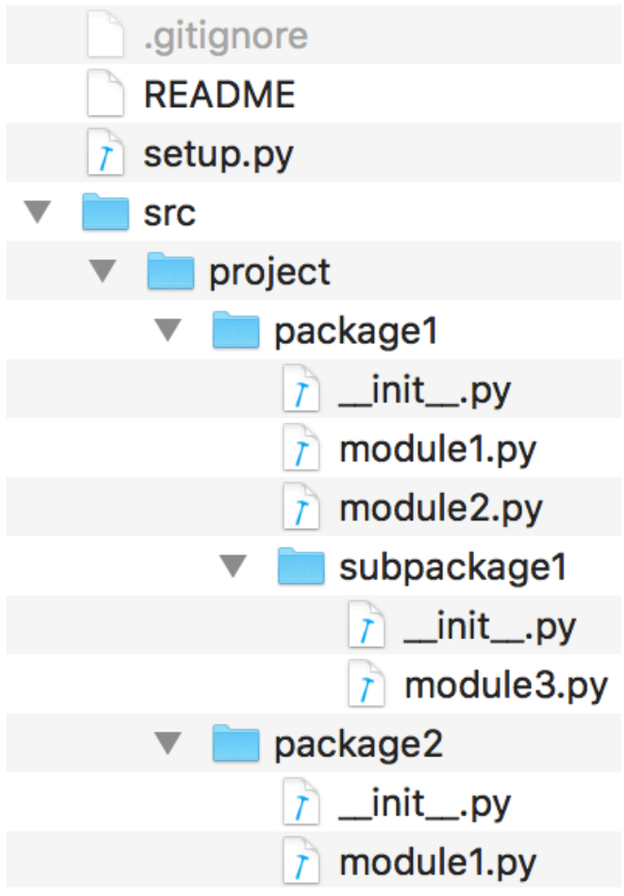
As we will soon see, this structure also helps with making our libraries installable.

- Add an `__author__` field to each file with the author's name/ID. This helps with knowing who to contact when questions/bugs arise with the relevant file.
- Add TODO items in the code using a comment line with format:

```
# TODO(author): This needs to be done.
```

Installable Packages

We often want to make our packages/libraries **installable** for distribution or for installing them on a production server. We can do that using the `setuptools` package. Simply add a `setup.py` script in the project's root directory:



Example / Template

```
1   import os
2   from setuptools import setup, find_packages
3
4
5   def read(filename):
6       return open(os.path.join(os.path.dirname(__file__), filename)).read()
7
8   setup(
9       name='project',
10      version='0.1dev',
11      license='Apache License 2.0',
12      packages=find_packages('src'),
13      package_dir={'': 'src'},
14      description='Example Project',
15      long_description=read('README.md'),
16      url='https://github.com/eaplatanios/example_project',
17      install_requires=[
18          'cython', 'numpy', 'pandas', 'ruamel.yaml', 'six', 'tensorflow',
19          'jnius==1.1-dev'],
20      extras_require={
21          'data': ['liac-arff', 'patool'],
22          'plotting': ['matplotlib']},
23      package_data={
24          'project': ['logging.yaml']}
25  )
```

package location mapping

package dependencies

References

- Official Python documentation at http://docs.python.org
- https://stackoverflow.com/questions/14132789/relative-imports-for-the-billionth-time