Recommended Project Layout

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham sixty-north.com

Overview

A project structure for most projects

Project description files

Extending packages with plugins

Different methods for implementing plugins

Python Project Structure

```
project_name/
    README.rst
    docs/
    src/
      package_name/
        __init__.py
        more_source.py
        subpackage1/
          __init__.py
    tests/
      test_code.py
    setup.py
```

◀ Project root - *not the package*

```
project_name/
    README.rst
    docs/
    src/
      package_name/
        __init__.py
        more_source.py
        subpackage1/
          __init__.py
    tests/
      test_code.py
    setup.py
```

- Project root - *not the package*
- Overview documentation

README.rst / reStructuredText

```
============
Project Name
============

A brief description of the project.

Section 1
=========

Installation or "quick start" information can go here.

Subsection
----------

Some details can go here.
```

```
project_name/
    README.rst
    docs/
    src/
      package_name/
        __init__.py
        more_source.py
        subpackage1/
          __init__.py
    tests/
      test_code.py
    setup.py
```

- Project root - *not the package*
- Overview documentation
- Project documentation

Documentation

Can come in many forms (e.g. Sphinx)

It should be easy to find

`README.rst` **should be in project root**

- Common convention
- Often refers to docs directory

```
project_name/
    README.rst
    docs/
    src/
      package_name/
        __init__.py
        more_source.py
        subpackage1/
          __init__.py
    tests/
      test_code.py
    setup.py
```

- Project root - *not the package*
- Overview documentation
- Project documentation
- Package/production code

src and sys.path

If this is in sys.path...

```
project_name/
    src/
      package_name/
        __init__.py
```

...then this is not importable

The `src` directory ensures that you develop against *installed versions* of your packages

```
project_name/
    README.rst
    docs/
    src/
      package_name/
        __init__.py
        more_source.py
        subpackage1/
          __init__.py
    tests/
      test_code.py
    setup.py
```

- Project root - *not the package*
- Overview documentation
- Project documentation
- Package/production code

```
project_name/
    README.rst
    docs/
    src/
      package_name/
        __init__.py
        more_source.py
        subpackage1/
          __init__.py
    tests/
      test_code.py
    setup.py
```

- Project root - *not the package*
- Overview documentation
- Project documentation
- Package/production code

- All tests for the project

Separation of Test and Production Code

Test and production code serve different purposes

Usually don’t want tests installed with package

Avoids tools treating tests as production code

Be pragmatic! Put tests in production code if necessary

A Simple, Practical Starting Point

```
project_name/
    README.rst
    docs/
    setup.py
    src/
      package_name/
        __init__.py
        more_source.py
        subpackage1/
          __init__.py
    tests/
      test_code.py
```

A Concrete Example: `demo_reader`

Extending Packages with Plugins

Packages define *extension points*

Extensions are implemented outside the package

Extensions are discovered at runtime

We'll look at two methods

- Namespace packages and `pkgutil`
- `setuptools` *entry points*

Implementing Plugins with Namespace Packages

Core package designates subpackages as extension points

Core package scans subpackages at runtime to discover plugins

Plugins augment the namespace package's extensible subpackages

```python
def iter_namespace(ns_pkg):
    return pkgutil.iter_modules(
        ns_pkg.__path__,
        ns_pkg.__name__ + ".")

compression_plugins = {
    importlib.import_module(
        module_name)
    for _, module_name, _
    in iter_namespace(
        demo_reader.compressed)
}


extension_map = {
    module.extension: module.opener
    for module in compression_plugins
}
```

- **Namespace package argument ns_pkg**
- **Finds all sub packages**

- **Ensure absolute package names**

- **Build set of module objects**
  - Import them with `importlib`

  - Find modules to import with `iter_namespace`

- **Build `extension_map` dict comprehension**
  - Look for module-level attributes
  - Get modules from `compression_plugins`

Implementing Plugins with `setuptools`

Implementing Plugins with `setuptools` Entry Points

Define extension points using `setuptools`

Plugins add to extension points in `setup.py`

Core package iterates over plugins added to extension points

```
compression_plugins = {
    entry_point.load()
    for entry_point
    in pkg_resources.iter_entry_points(
        'demo_reader.compression_plugins')
}

extension_map = {
    module.extension: module.opener
    for module in compression_plugins
}
```

- **Build set of modules**
  - `load()` returns a module object in this case
  - Iterate over all extensions to the entry point

`extension_map` definition unchanged

Defining Entry Points

```
setuptools.setup(
  entry_points={
    'demo_reader.compression_plugins': [
      'bz2 = demo_reader_bz2.bzipped'
    ]
  }
)
```

Entry point name

Extension name

Extension object

Summary

A project structure that supports all aspects of code construction

Separating production and test code

Install packages into a Python environment

Use plugins to extend packages

- Namespace packages
- `setuptools`