



Python projects' Best Practices





Who am I

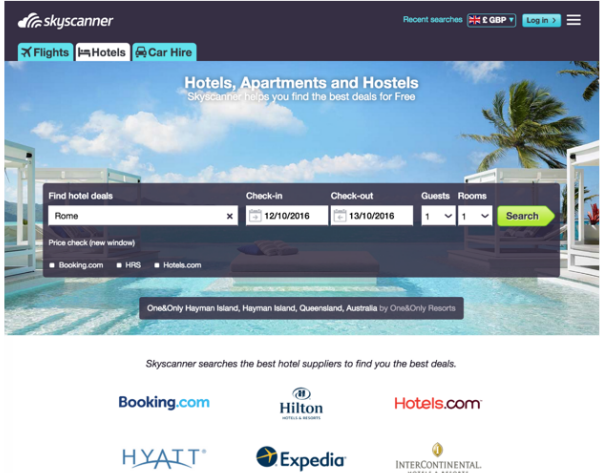
Enrique Saez

Software engineer working for Skyscanner Hotels: <http://skyscanner.net/hotels>

Motivation

- Started in a mature Python project
- CI & CD already set up





IntegrationBuild (Linux) ▾			Pending (2) ▾	Run
#7928	Tests failed: 2 (1 new), passed: 1897, ignored: 31; Running '[TeamCity] Server tests' ▾	No artifacts ▾	Changes (3) ▾	1h:58m left
#7927	Tests failed: 1 (1 new), passed: 6955, ignored: 55, muted: 3; Running '[TeamCity] Server tests' ▾	No artifacts ▾	Evgeniy Koshkin (1) ▾	1h:19m left
#7926	Tests passed: 12149, ignored: 75, muted: 4; Running '[TeamCity] Integration tests' ▾	No artifacts ▾	Changes (4) ▾	40m:36s left
#7925	Tests failed: 1 (1 new), passed: 12303, ignored: 75, muted: 3; Running '[TeamCity] Integration tests...' ▾	No artifacts ▾	Changes (2) ▾	overtime: 9m:15s
#7924	Tests failed: 2 (1 new), passed: 13705, ignored: 91, muted: 4 ▾	No artifacts ▾	Changes (3) ▾	10 hours ago (2h:02m)

Goal

- Help advanced beginners understand project structure

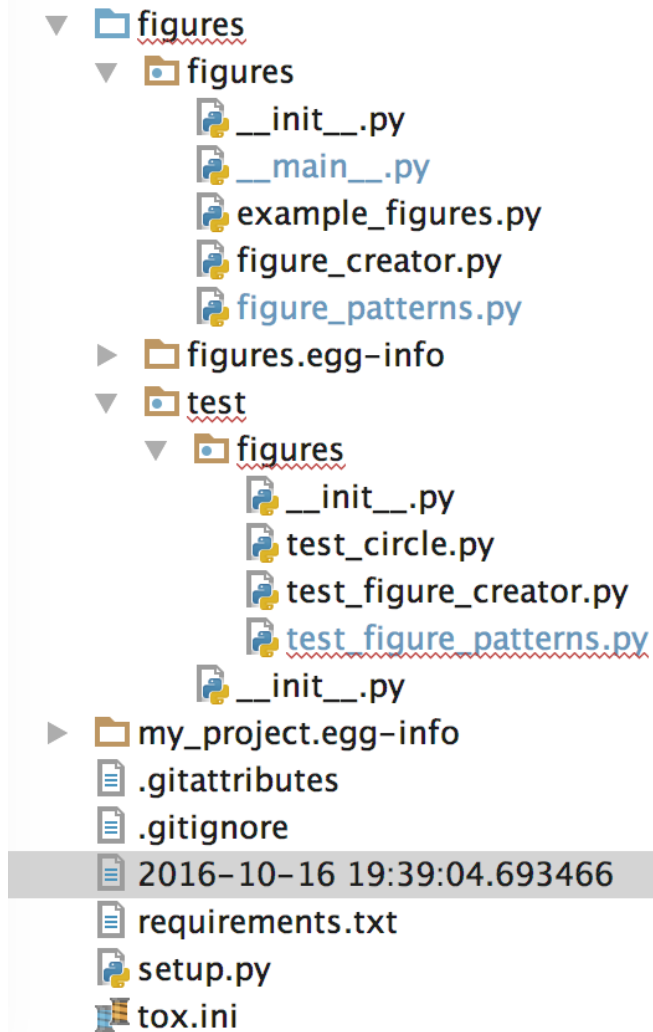
Best Practices for a Python project

- Project structure
- Modules
- Packages
- PEP8
- Environment setup (virtualenv)
- Testing
- Distributing your project

Project structure

project folder	Flask	Requests	boto3	pandas
docs/	docs/			
examples	examples/			
sample/core.py	flask/	requests/	boto3/	pandas/
scripts/	scripts/		scripts/	scripts/
tests/	tests/	tests/	tests/	/pandas/tests/
Makefile	Makefile	Makefile	Makefile	Makefile
setup.py	setup.py	setup.py	setup.py	setup.py
requirements.txt	setup.cfg	requirements.txt	setup.cfg	requirements.txt
tox.ini	tox.ini		tox.ini	tox.ini

Example project structure: Figures



Modules

- split code in different files for related data and functionality
- lowercase, _separated names for module and function names: create_square

```
def create_square(start, stop):  
    print i**2  
square(0, 10)
```

- square(0,10) will get run on import!

```
def create_square(start, stop):  
    print i**2  
if __name__ == '__main__':  
    square(0, 10)
```


What does python do to import a module?

- Check the module registry (`sys.modules`)
- If the module is already imported:
 - Python uses the existing module object as is
- Otherwise:
 1. Create a new, empty module object (essentially a dictionary)
 2. Insert that module object in `sys.modules` dictionary
 3. Load the module code object (if necessary, compile the module first)
 4. Execute the module code object in the new module's namespace (isolated scope)
 5. Top-level statements in `modu.py` will be executed, including other imports
- It's fairly cheap to import an already imported module: look the module name up in a dictionary. $O(1)$

Importing a module (II)

- Function and class definitions are stored in the module's dictionary
 - Available to the caller through the module's namespace
 - The included code is isolated in a module namespace:
 - Generally don't have to worry about the included code having unwanted effects (overriding functions with the same name)

Packages

```
pack/  
pack/__init__.py  
pack/modu.py
```

```
python setup.py install
```

Installed into `/dist-packages/`

Don't have to worry about configuring PYTHONPATH to include the source

Packages (II)

```
sound/__init__.py  
sound/effects/__init__.py  
sound/effects/echo.py  
sound/effects/surround.py
```

```
from sound.effects import surround    import sound.effects.surround as surround
```

- Execute all top-level statements from `__init__.py`
- Execute all top-level statements from `surround.py`
- Any `public` variable, function, class defined in `surround.py` is available in `sound.effects.surround`

PEP8 (Style Guide for Python code)

Improve the readability of code and make it consistent

- Four spaces (NOT a tab) for each indentation level
- Limit all lines to 80/120 characters
- Separate:
 - top level functions and class definitions with 2 blank lines
 - methods inside a class by a single blank line

```
from figures.figures.figure_patterns import FigurePatterns
```

```
class CircleCreator(FigurePatterns, object):  
  
    def __init__(self, name, area=7):  
        super(CircleCreator, self).__init__(name)  
        self.area = area
```

PEP8 (II)

- Lowercase, _-separated names for module and function names: `my_module`
- CamelCase to name classes
- `'_'` prefix: "private" variable/method not to be used outside the module
- blank spaces, CONSTANTS

```
from figures.figures.figure_patterns import FigurePatterns
```

```
class CircleCreator(FigurePatterns, object):
```

```
    LINE_WIDTH = 5
```

```
    def _compute_area(self):  
        return random.random()*10
```

PEP8 (III)

- imports:
 - standard
 - third-party
 - local library

```
from collections import defaultdict
from requests import
from figures import figure_patterns
```

Testing: environment setup (virtualenv)

- Tool to create isolated Python environments
 - Python packages installed in an isolated location rather than globally.
 - Keep dependencies separated
 - Isolated environments with different python versions

virtualenv

```
$ virtualenv venv  
$ virtualenv -p /usr/bin/python2.7 venv  
$ source venv/bin/activate  
$ deactivate  
$ pip freeze > requirements.txt (list packages and version in venv)  
$ pip install -r requirements.txt
```

- **Creates:**
 - a folder containing the necessary executables to use the packages needed by the Python project
 - a copy of pip to install other packages

testing: (unittest package)

- Mirror hierarchy:

mylib/foo/bar.py

mylib/tests/foo/test_bar.py

```
from unittest import TestCase
```

```
class TestFigures(TestCase):
```

```
    def setUp(self):
```

```
        self.circle = CircleCreator('Circle')
```

```
    def tearDown(self):
```

```
        self.circle = None
```

```
    def test_name_ok(self):
```

```
        self.assertEqual(self.circle.get_name(), 'Circle')
```

- assert method provided by unittest

testing: Fixtures

Resources/initial conditions that a test needs to operate correctly and independently from other tests.

Functions and methods that run before and after a test

```
from unittest import TestCase
```

```
class TestFigures(TestCase):
```

```
    def setUp(self):  
        self.circle = CircleCreator('Circle')
```

```
    def tearDown(self):  
        self.circle = None
```

```
    def test_name_ok(self):  
        self.assertEqual(self.circle.get_name(), 'Circle')
```

testing: (nose package)

- Provides automatic test discovery
- Loads every file that starts with test_
- Executes all functions within that start with test_
- In maintenance mode for the past several years: use Nose2, py.test

```
$ nosetest
```

test selection:

```
$ path.to.your.module:ClassOfYourTest.test_method
```

```
$ path.to.your.module:ClassOfYourTest
```

```
$ path.to.your.module
```

py.test

- Auto-discovery of test modules and functions
- Modular fixtures for managing small or parametrized test resources
- Can run unittest and nose test suites

```
$ py.test tests/
```

tox

- Clean environment for running unit tests
- Create virtual environment, using pip to install dependencies
- Use setup.py to install package inside virtualenv
- Run tests
- Automate and standardize how tests are run in Python for each environment

```
[tox]
```

```
envlist = {py27}
```

```
[testenv]
```

```
deps =
```

```
    -rrequirements.txt
```

```
commands =
```

```
    nosetests figures/test/
```

Jargon

- **Built Distribution**
 - A Distribution format containing files and metadata
 - Only need to be moved to the correct location to be installed
- **Source Distribution (or “sdist”)**
 - requires a build step when installed by pip
 - provides metadata and the essential source files needed for pip, or generating a Built Distribution.
 - usually generated with `setup.py sdist`
 - see the `bdist_wheel` setuptools extension available from the wheel project to create wheels
- **setuptools**
 - Collection of enhancements to the Python distutils, (includes `easy_install`)
 - Easily build and distribute Python distributions, especially ones that have dependencies on other packages.

Jargon (II)

- **pip**
 - The PyPA recommended tool for installing Python packages
- **Wheel**
 - A Built Distribution format supported by pip
- **egg**
 - a zip file with different extension
- **setup.cfg**
 - ini file that contains option defaults for setup.py commands.

setup.py

```
from setuptools import setup, find_packages

setup(
    name="figures",
    version="1",
    description="figures module to create your own figures",
    packages=packages=['figures'],
    package_dir = {'': 'figures'},
    entry_points={
        'console_scripts': [
            "figures = figures.example_figures:main",
        ],
    },
)
```

- entry points: package.subpackage:function

setup.py (II)

- **Console scripts**
 - Installs a tiny program in the system path to call a module's specific function
 - Launchable programs need to be installed inside a directory in the systempath
- **entry points**
 - Part of setuptools
 - Used by other python programs to dynamically discover features that a package provides
 - entry_point_inspector package: lists the entry points available in a package

setup.py (III)

`python setup.py install`

will create a script like this in `/bin/`:

```
__requires__ = 'figures==1'
import sys
from pkg_resources import load_entry_point

if __name__ == '__main__':
    sys.exit(
        load_entry_point('figures==1', 'console_scripts', 'figure_creator')()
    )
```

- scans the entry points of the figures package
- retrieves the figures key from the console_scripts category

Requirements for Installing Packages

- pip, setuptools (for advanced installations) and wheel
- distutils for simple package installations
- Create a virtual environment
- pip

```
$ pip install -r requirements.txt
```

```
$ pip install 'botocore=0.6.8'
```

Wheel

- pre-built distribution format
- **faster installation compared to Source Distributions (sdist)**
especially if project contains compiled extensions
- zip file with a different extension
- Better caching for testing and continuous integration
- Wheel files do not require installation

Wheel (II)

- supported by pip
- Offers the bdist_wheel setuptools extension for creating wheel distributions
- Command line utility for creating and installing wheels

```
python setup.py bdist_wheel
```

- creates a .whl file in the /dist/ directory