Jeff Knupp

PYTHON PROGRAMMER

Everything I know about Python...

> Discuss Posts With Other Readers at **discourse.jeffknupp.com (http://discourse.jeffknupp.com)**!

Open Sourcing a Python Project the Right Way (/blog/2013/08/16/open-sourcing-a-python-project-the-right-way)

Most Python developers have written at least *one* tool, script, library or framework that others would find useful. My goal in this article is to make the process of open-sourcing existing Python code as clear and painless as possible. And I don't simply mean, "create GitHub repo, `git push`, post on Reddit, and call it a day." By the end of this article, you'll be able to take an existing code base and transform it into an open source project that encourages both use *and* contribution.

While every project is different, there are some parts of the process of open-sourcing existing code that are common to *all* Python projects. In the vein of another popular series I've written, "Starting a Django Project The Right Way," (http://www.jeffknupp.com/blog/2012/10/24/starting-a-django-14-project-the-right-way/) I'll outline the steps I've found to be necessary when open-sourcing a Python project.

Update (Aug 17): Thanks to @pydanny (http://www.twitter.com/pydanny) for alerting me about the existence of Cookiecutter (https://github.com/audreyr/cookiecutter-pypackage), an awesome project by @audreyr (https://twitter.com/audreyr). I've added a section on it to the end of this article. Be sure to check out Audrey's awesome project!

Update 2 (Aug 18): Thanks to @ChristianHeimes (http://www.twitter.com/ChristianHeimes) (and others) for suggesting a section on `tox` . Christian also reminded me about PEP 440 and had some great suggestions for other minor improvements, all of which have been implemented.

Tools and Concepts

In particular, there are a number of tools and concepts I've found useful or necessary. I'll cover each of the topics below, including the precise commands you'll need to run and configuration values you'll need to set. The goal is to make the entire process clear and simple.

1. Project layout (directory structure)
2. `setuptools` and the `setup.py` file
3. git (http://www.git-scm.com) for version control
4. GitHub (http://www.github.com) for project management
    1. GitHub's "Issues" for the following:
        1. bug tracking
        2. feature requests
        3. planned features
        4. release/version management
5. git-flow (http://nvie.com/posts/a-successful-git-branching-model/) for git workflow
6. py.test (http://www.pytest.org) for unit testing
7. tox (http://tox.readthedocs.org/en/latest/) for testing standardization
8. Sphinx (http://www.sphinx-doc.org) for auto-generated HTML documentation
9. TravisCI (https://travis-ci.org/) for continuous testing integration
10. ReadTheDocs (https://readthedocs.org) for continuous documentation integration
11. Cookiecutter (https://github.com/audreyr/cookiecutter-pypackage) to automate these steps when starting your next project

Project Layout

When setting up a project, the *layout* (or *directory structure*) is important to get right. A sensible layout means that potential contributors don't have to spend forever hunting for a piece of code; file locations are intuitive. Since we're dealing with an existing project, it means you'll probably need to move some stuff around.

Let's start at the top. Most projects have a number of top-level files (like `setup.py`, `README.md`, `requirements.txt`, etc). There are then three directories that *every* project should have:

1. A `docs` directory containing project documentation
2. A directory named with the project's name which stores the actual Python package
3. A `test` directory in one of two places
    1. Under the package directory containing test code and resources
    2. As a stand-alone top level directory

To get a better sense of how your files should be organized, here's a simplified snapshot of the layout for one of my projects, sandman (http://www.github.com/jeffknupp/sandman):

```
$ pwd
~/code/sandman
$ tree
.
|- LICENSE
|- README.md
|- TODO.md
|- docs
|   |-- conf.py
|   |-- generated
|   |-- index.rst
|   |-- installation.rst
|   |-- modules.rst
|   |-- quickstart.rst
|   |-- sandman.rst
|- requirements.txt
|- sandman
|   |-- __init__.py
|   |-- exception.py
|   |-- model.py
|   |-- sandman.py
|   |-- test
|       |-- models.py
|       |-- test_sandman.py
|- setup.py
```

As you can see, there are some top level files, a `docs` directory (`generated` is an empty directory where sphinx will put the generated documentation), a `sandman` directory, and a `test` directory under `sandman`.

`setuptools` and the `setup.py` File

The `setup.py` file you've likely seen in other packages is used by the `distutils` package for the installation of Python packages. It's an important file for any project, as it contains information on versioning, package requirements, the project description that will be used on PyPI, and your name and contact information, among many other things. It allows packages to be searched for and installed in a programmatic way, providing meta-data and instructions to tools that do so.

The `setuptools` (https://pythonhosted.org/setuptools/setuptools.html) package (really a set of enhancements for `distutils`) simplifies the building and distribution of Python packages. A Python package that was packaged with `setuptools` should be indistinguishable from one packaged with `distutils`. There's really no reason not to use it.

`setup.py` should live in your project's root directory. The most important section of `setup.py` is the call to `setuptools.setup`, where all the meta-information about the package lives. Here's the complete contents of `setup.py` from sandman (http://www.github.com/jeffknupp/sandman):

```
from __future__ import print_function
from setuptools import setup, find_packages
from setuptools.command.test import test as TestCommand
import io
import codecs
import os
import sys

import sandman

here = os.path.abspath(os.path.dirname(__file__))

def read(*filenames, **kwargs):
    encoding = kwargs.get('encoding', 'utf-8')
    sep = kwargs.get('sep', '\n')
    buf = []
    for filename in filenames:
        with io.open(filename, encoding=encoding) as f:
            buf.append(f.read())
    return sep.join(buf)

long_description = read('README.txt', 'CHANGES.txt')

class PyTest(TestCommand):
    def finalize_options(self):
        TestCommand.finalize_options(self)
        self.test_args = []
        self.test_suite = True

    def run_tests(self):
        import pytest
        errcode = pytest.main(self.test_args)
        sys.exit(errcode)

setup(
    name='sandman',
    version=sandman.__version__,
    url='http://github.com/jeffknupp/sandman/',
    license='Apache Software License',
    author='Jeff Knupp',
    tests_require=['pytest'],
    install_requires=['Flask>=0.10.1',
                    'Flask-SQLAlchemy>=1.0',
                    'SQLAlchemy==0.8.2',
                    ],
    cmdclass={'test': PyTest},
    author_email='jeff@jeffknupp.com',
    description='Automated REST APIs for existing database-driven systems',
    long_description=long_description,
    packages=['sandman'],
    include_package_data=True,
    platforms='any',
    test_suite='sandman.test.test_sandman',
    classifiers = [
        'Programming Language :: Python',
        'Development Status :: 4 - Beta',
        'Natural Language :: English',
        'Environment :: Web Environment',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: Apache Software License',
        'Operating System :: OS Independent',
        'Topic :: Software Development :: Libraries :: Python Modules',
        'Topic :: Software Development :: Libraries :: Application Frameworks',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
        ],
    extras_require={
        'testing': ['pytest'],
    }
)
```

(thanks to Christian Heimes for the suggestion to make `read` more idiomatic. I'll in turn let whichever project I stole this code from know...)

Most of the contents are straightforward and could be gleaned from the `setuptools` documentation, so I'll only touch on the "interesting" parts. Using `sandman.__version__` and the method of getting `long_description` (taken from the `setup.py` of other projects, though I can't remember which ones) reduce the amount of boilerplate code we need to write. Instead of maintaining the project's version in three places (`setup.py` , the package itself via `package.__version__` , and the documentation), we can always use the package's version to populate the `version` parameter in `setup` .

`long_description` is the document used by PyPI as the description on your project's PyPI page. As there is another file, `README.md` with almost the exact same content, I use pandoc (http://johnmacfarlane.net/pandoc/) to automatically generate `README.rst` from `README.md`. Thus, we can simply `read` the file `README.rst` and use that as the value for `long_description`.

py.test (discussed below) has a special entry (`class PyTest`) to allow `python setup.py test` to work correctly. That code snippet was taken directly from the `py.test` documentation.

Everything else is in the file is simply setting values for the `setup` parameters described in the documentation.

Other `setup.py` parameters

There are some `setup` arguments that sandman (http://www.github.com/jeffknupp/sandman) has no use for, but your package might. For example, you may be distributing a script that you'd like your user to be able to execute from the command line. In the example above, that script would only be installed in the normal `site-packages` location along with the rest of your code. There would be no (easy) way for the user to run it after it was installed.

For that reason, `setup` can take a `scripts` argument that specifies Python scripts that should be installed as such. To install a script called `go_foo.py` from your package, the call to `setup` would include the line:

```
scripts = ['go_foo.py'],
```

Just make sure you put the relative path to your script, not just its name (e.g. `scripts = ['scripts/foo_scripts/go_foo.py']`). Also, your script should begin with a "shebang" line with "python" in it, like:

```
 #! /usr/bin/env python
```

`distutils` will automatically replace this line with the current interpreter location during installation.

If your package is more complex than the simple one discussed here, take a look at both the `setuptools` (https://pythonhosted.org/setuptools/setuptools.html) documentation and "Distributing Python Modules" (http://docs.python.org/2/distutils/index.html) from the official documentation. Between the two, you should be able to straighten out any issues you might have encountered.

Source Control With Git, Project Management with GitHub

In "Starting a Django Project The Right Way," (http://www.jeffknupp.com/blog/2012/10/24/starting-a-django-14-project-the-right-way/) I suggest either git or mercurial for version control. For a project meant to be both shared and contributed to, there's really only one choice: git. In fact, I'll go so far as to say that not only is the use of git necessary, you'll also need to use GitHub (http://www.github.com) to maintain your project if you want people to actually use and contribute to it.

It's not meant to be an inflammatory statement (though no doubt many will take issue with it). Rather, for better or worse, git and GitHub (http://www.github.com) have become the de-facto standard for Open Source projects. GitHub is the site potential contributors are most likely to be registered on and familiar with. That, I believe, is not a point to be taken lightly.

Create a `README.md` File

The project description for repos on GitHub is taken from a file in the project's root directory: `README.md`. This file should contain the following pieces of information:

- A description of your project
- Links to the project's ReadTheDocs page
- A TravisCI button showing the state of the build
- "Quickstart" documentation (how to quickly install and use your project)
- A list of non-Python dependencies (if any) and how to install them

It may sound silly, but this is an important file. It's quite likely to be the first thing both prospective users *and* contributors read about your project. Take some time to write a clear description and make use of GFM (**G**itHub**F**lavored**M**arkdown) to make it look somewhat attractive. You can actually create/edit this file right on GitHub with a live-preview editor if you're not comfortable writing documents in raw Markdown.

We haven't yet covered the second and third items in the list yet (ReadTheDocs and TravisCI). You'll find these discussed below.

Using the "Issues" Page

Like most things in life, the more you put into GitHub, the more you get out of it. Since users will be using it to file bug reports anyway, making use of GitHub's "Issues" page to track feature requests and enhancements just makes sense.

More importantly, it allows potential contributors to both see a list of things they might implement and automatically manages the pull request workflow in a reasonably elegant manner. GitHub issues and their comments can be cross-linked with commits, other issues in your project, issues in *other* projects, etc. This makes the "Issues" page a good place to keep all of the information related to bug fixes, enhancements, and feature requests.

Make sure to keep "Issues" up to date and to at least briefly respond to new issues in a timely manner. As a contributor, there's nothing more demotivating than fixing a bug and watching as it languishes on the issues page, waiting to be merged.

A Sensible git Workflow With git-flow

To make things easier on both yourself and contributors, I suggest using the very popular git-flow (http://nvie.com/posts/a-successful-git-branching-model/) model of branching.

Quick Overview

The `develop` is the branch you'll be doing most of your work off of; it's also the branch that represents the code to be deployed in the next release. `feature` branches represent non-trivial features and fixes that have not yet been deployed (a completed `feature` branch is merged back into `develop`). Updating `master` is done through the creation of a `release`.

Installation

Install git-flow by following the instructions for your platform here (https://github.com/nvie/gitflow/wiki/Installation).

Once installed, you can migrate your existing project with the command

```
$ git flow init
```

Branch Details

You'll be asked a number of configuration questions by the script. The default values suggested by git-flow are fine to use. You may notice your default branch is set to `develop`. More on that in a moment. Let's take a step back and describe the git-flow... erm, flow, in a bit more detail. The easiest way to do so is to discuss the various branches and *types* of branches in the model.

Master

`master` is always "production ready" code. Commits are never made directly to `master`. Rather, code on `master` only gets there after a production release branch is created and "finished" (more on that in a sec). Thus the code on `master` is always able to be released to production. Also, `master` is always in a predictable state, so you never need to worry if `master` (and thus production) has changes one of your other branches doesn't.

Develop

Most of your work is done on the `develop` branch. This branch contains all of the completed features and bug fixes yet to be released; nightly builds or continuous integration servers should target `develop`, as it represents the code that will be included in the next release.

For one-off commits, feel free to commit to `develop` directly.

Feature

For larger features, a `feature` branch should be created. `feature` branches are created off of `develop`. They can be small enhancements for the next release or further out changes that, nonetheless, need to be worked on now. To start work on a new feature, use:

```
$ git flow feature start <feature name>
```

This creates a new branch: `feature/<feature name>`. Commits are then made to this branch as normal. When the feature is complete *and ready to be released to production*, it should be merged back into develop using the following command:

```
$ git flow feature finish <feature name>
```

This merges the code into `develop` and deletes the `feature/<feature name>` branch.

Release

A `release` branch is created from `develop` when you're ready to begin a production release. Create one using the following command:

```
$ git flow release start <release number>
```

Note that this is the first time a version number for the release is created. All completed and ready to be released features must already be on `develop` (and thus `feature finish`'ed). After your release branch is created, release your code. Any small bug fixes needed after the release are made directly to the `release/<release number>` branch. Once it has settled down and no more bug fixes seem necessary, run the following command:

```
$ git flow release finish <release number>
```

This merges your `release/<release number>` changes back into both `master` *and* `develop`, meaning you never need to worry about either of those branches lacking changes that are in production (perhaps as the result of a quick bug fix).

Hotfix

While potentially useful, `hotfix` branches are, I would guess, little used in the real world. A `hotfix` is like a `feature` branch off of `master`: if you've already closed a `release` branch but realize there are vital changes that need to be released, create a `hotfix` branch off of `master` (at the tag created during `$ git flow release finish <release number>`) like so:

```
$ git flow hotfix start <release number>
```

After you make your changes and bump your version number, finalize the `hotfix` via

```
$ git flow hotfix finish <release number>
```

This, like a `release` branch (since it essentially *is* a type of release branch), commits the changes to both `master` and `develop`.

The reason I assume they're rarely used is because there is already a mechanism for making changes to released code: committing to an un-`finish`ed release branch. Sure, in the beginning, teams may `git flow release finish ...` too early, only to find they need to make some quick changes the next day. Over time, though, they'll settle on a reasonable amount of time for a `release` branch to remain open and, thus, won't have a need for `hotfix` branches. The only *other* time you would need a `hotfix` branch is if you needed a new "feature" in production immediately, without picking up the changes already in `develop`. That strikes me as something that happens (hopefully) very rarely.

`virtualenv` and `virtualenvwrapper`

Ian Bicking's `virtualenv` tool has become the de-facto standard mechanism for isolating Python environments. Its purpose is simple: if you have a number of Python projects on a single machine, each with different dependencies (perhaps with dependencies on different versions of the same package), managing the dependencies in a single Python installation is nigh impossible.

`virtualenv` creates "virtual" Python installations, each with their own, segregated, `site-packages`. `distribute` and `pip` are also installed in such a way that `pip install` correctly installs packages to the `virtualenv` rather than the system Python installation. Switching back and forth between your `virtualenv` is a one-command process.

A separate tool, Doug Hellmann's `virtualenvwrapper`, makes creating and managing multiple `virtualenv`s easier. Let's go ahead and install both now:

```
$ pip install `virtualenvwrapper`
...
Successfully installed `virtualenvwrapper` `virtualenv` `virtualenv`-clone stevedore
Cleaning up...
```

As you can see, the latter has a dependency on the former, so simply installing `virtualenvwrapper` is sufficient. Note that if you're using Python 3, PEP-405 (http://www.python.org/dev/peps/pep-0405/), which gives Python native support for virtual environments through the `venv` package and `pyvenv` command, was implemented in Python 3.3. You should use that instead of the tools mentioned above.

Once you've installed `virtualenvwrapper`, you'll need to add a line to your `.zhsrc` file (or `.bashrc` file for bash users):

```
$ echo "source /usr/local/bin/virtualenvwrapper.sh" >> ~/.zshrc
```

This adds a number of useful commands to your shell (remember to `source` your `.zshrc` to actually make them available for the first time). While you can create a `virtualenv` directly with the `mkvirtualenv` command, creating a "*project*" using `mkproject [OPTIONS] DEST_DIR` is usually more useful. Since we have an existing project, however, we'll simply create a new `virtualenv` for our project. We can do this with a simple command:

```
$ mkvirtualenv ossproject

New python executable in ossproject/bin/python
Installing setuptools............done.
Installing pip...............done.
(ossproject)$
```

You'll notice your shell prompt is now prepended by the name of your `virtualenv` (which I called "ossproject", but obviously you can use whatever name you'd like). Now anything installed via `pip install` is installed to the `site-packages` of your `virtualenv`.

To stop working on your project and switch back to the system installation, use the `deactivate` command. You should see the `virtualenv` name that was prepended to your shell prompt disappear. To resume work on your project, run `$ workon <project name>` and you'll be back in your `virtualenv`.

Aside from simply creating the `virtualenv` for your project, you'll use it to do one more thing: generate your `requirements.txt` file. `pip` is capable of installing all of project's dependencies by using a requirements file and the `-r` flag. To create this file, run the following command within your `virtualenv` (once your code is working with the `virtualenv`, that is):

```
(ossproject)$ pip freeze > requirements.txt
```

You'll get a nice list of all of the requirements for your project, which can later be used by the setup.py file to list your dependencies. One note here: I often change the '==' to '>=' in `requirements.txt` to say "any version of this package after the one I'm working on." Whether or not you should/need to do this is project specific, but I just thought I'd point it out.

Commit `requirements.txt` to your git repo. In addition, you can now add the packages listed there as the value for the `install_requirements` argument to `distutils.setup` in `setup.py`. Doing that now will ensure that, when we later upload the package to PyPI. It can be `pip install`ed with automatically resolved dependencies.

Testing With py.test

In the Python automated testing ecosystem, there are two main alternatives to the (quite usable) Python standard library `unittest` package: nose (http://www.nosetest.org) and py.test (http://www.pytest.org). Both extend `unittest` to make it easier to work with while adding additional functionality. Truthfully, either is a fine choice. I happen to prefer `py.test` for a few reasons:

- Support for setuptools/distutils projects
  - `python setup.py test` still works
- Support for "normal" `assert` statements (rather than needing to remember all the jUnit-style assert functions)
- Less boilerplate
- Support for multiple testing styles
  - `unittest`
  - `doctest`
  - nose tests

Note

If you already have an automated testing solution, feel free to continue using it and skip this section. Be warned that later sections may assume testing is done using py.test, which may affect configuration values.

Test Setup

In the `test` directory, wherever you decided it should live, create a file called `test_<project_name>.py`. py.test's test discovery mechanism will treat any file with the `test_` prefix as a test file (unless told otherwise).

What you put in that file is largely up to you. Writing tests is a giant topic and outside of the scope of this article. The important thing, however, is that the tests are useful to both you *and potential contributors*. It should be clear what functionality each test is exercising. Tests should be written in the same "style" so that a potential contributor doesn't have to guess which of the three styles of testing used in your project he/she should use.

Test Coverage

Automated test coverage is a contentious topic. Some believe it to be a meaningless metric that gives false security. Others find it genuinely useful. At the very least, I would suggest if you already have tests and have *never* checked your test coverage, do so now as an exercise.

With py.test, we can make use of Ned Batchelder's coverage (http://nedbatchelder.com/code/coverage/) tool. To do so, `$ pip install pytest-cov` . If you previously ran your tests like this:

```
$ py.test
```

you can generate test coverage reports by passing a few additional flags. Below is an example of running `sandman`

```
$ py.test --cov=path/to/package
$ py.test --cov=path/to/package --cov-report=term --cov-report=html
====================================================== test session starts ===========
============================================
platform darwin -- Python 2.7.5 -- pytest-2.3.5
plugins: cov
collected 23 items

sandman/test/test_sandman.py .......................
---------------------------------------- coverage: platform darwin, python 2.7.5-final
-0 -----------------------------------------
Name                           Stmts   Miss  Cover
--------------------------------------------------
sandman/__init__                   5      0   100%
sandman/exception                 10      0   100%
sandman/model                     48      0   100%
sandman/sandman                  142      0   100%
sandman/test/__init__              0      0   100%
sandman/test/models               29      0   100%
sandman/test/test_sandman        114      0   100%
--------------------------------------------------
TOTAL                            348      0   100%
Coverage HTML written to dir htmlcov

=================================================== 23 passed in 1.14 seconds ========
===================================================
```

Certainly not all of my projects have 100% test coverage (in fact, as you read this, `sandman` might not have 100% coverage anymore). Getting to 100% was a useful exercise, though. It exposed bugs and opportunities for refactoring I wouldn't have otherwise noticed.

Since, as for the tests themselves, test coverage reports can be generated automatically as part of your continuous integration. If you choose to do so, displaying a badge showing your current test coverage adds a bit of transparency to your project (and high numbers can sometimes encourage others to contribute).

Standardized Testing With Tox

One issue all Python project maintainers face is *compatibility*. If your goal is to support both Python 2.x and Python 3.x (and, if you currently only support Python 2.x, it should be), how do you make sure your project actually works against all the versions you say you support? After all, when you run your tests, you're only testing the specific interpreter version used to run the tests. It's quite possible that a change you made works fine in Python 2.7.5 but breaks in 2.6 and 3.3.

Luckily, there's a tool dedicated to solving this exact problem. tox (http://tox.readthedocs.org/en/latest/) provides "standardized testing in Python," and it goes beyond merely running your tests with more than one version of the interpreter. It creates a fully sandboxed environment in which your package and its requirements are installed and tested. If you made a change that works fine when tested directly but the change inadvertently broke your *installation*, you'll discover that with tox.

`tox` is configured via an `.ini` file: `tox.ini`. It's a very simple file to set up. Here's a minimal `tox.ini` file taken from the tox documentation:

```
# content of: tox.ini , put in same dir as setup.py
[tox]
envlist = py26,py27
[testenv]
deps=pytest       # install pytest in the venvs
commands=py.test  # or 'nosetests' or ...
```

By setting `py26` and `py27` in the `envlist`, `tox` knows that it should run your tests against those versions of the interpreter. There are about a dozen "default" environments that `tox` supports out of the box, including `jython` and `pypy`. `tox` makes testing against different versions and configurations it would be a crime *not* to support multiple versions, if only to get to use such an awesome tool.

`deps` is a list of dependencies for your package. You can even tell `tox` to install all or some of your dependencies from an alternate PyPI URL. Clearly, quite a bit of thought and work has gone into the project.

Actually running your all of your tests against all of your environments now takes four keystrokes:

```
$ tox
```

A more complicated setup

My book, "Writing Idiomatic Python" (http://www.jeffknupp.com/writing-idiomatic-python-ebook/), is actually written as a series of Python modules and docstrings. This is done to make sure all the code samples work as intended. As part of my build process, I run `tox` to make sure the code in any new idioms works correctly. I also occasionally check my test coverage to make sure there are no idioms inadvertently being skipped during testing. As such, my `tox.ini` is a bit more complicated than the one above. Take a look:

```
[tox]
envlist=py27, py34

[testenv]
deps=
    pytest
    coverage
    pytest-cov
setenv=
    PYTHONWARNINGS=all

[pytest]
adopts=--doctest-modules
python_files=*.py
python_functions=test_
norecursedirs=.tox .git

[testenv:py27]
commands=
    py.test --doctest-module

[testenv:py34]
commands=
    py.test --doctest-module

[testenv:py27verbose]
basepython=python
commands=
    py.test --doctest-module --cov=. --cov-report term

[testenv:py34verbose]
basepython=python3.4
commands=
    py.test --doctest-module --cov=. --cov-report term
```

Even this config file is pretty straightforward. And the result?

```
(idiom)~/c/g/idiom git:master >>> tox
GLOB sdist-make: /home/jeff/code/github_code/idiom/setup.py
py27 inst-nodeps: /home/jeff/code/github_code/idiom/.tox/dist/Writing Idiomatic Python
-1.0.zip
py27 runtests: commands[0] | py.test --doctest-module
/home/jeff/code/github_code/idiom/.tox/py27/lib/python2.7/site-packages/_pytest/assert
ion/oldinterpret.py:3: DeprecationWarning: The compiler package is deprecated and remo
ved in Python 3.x.
from compiler import parse, ast, pycodegen
=============================================================== test session starts ==
==============================================================
platform linux2 -- Python 2.7.5 -- pytest-2.3.5
plugins: cov
collected 150 items
...
============================================================ 150 passed in 0.44 second
s ============================================================
py33 inst-nodeps: /home/jeff/code/github_code/idiom/.tox/dist/Writing Idiomatic Python
-1.0.zip
py33 runtests: commands[0] | py.test --doctest-module
=============================================================== test session starts ==
==============================================================
platform linux -- Python 3.3.2 -- pytest-2.3.5
plugins: cov
collected 150 items
...
============================================================ 150 passed in 0.62 second
s ============================================================
_____________________________________________________________________ summary ________
______________________________________________________________
py27: commands succeeded
py33: commands succeeded
congratulations :)
```

(I cut out the list of all the tests it runs from the output). If I want to see the coverage of my tests for an environment, I simply run:

```
$ tox -e py33verbose
-------------------------------------------------- coverage: platform linux, python 3.
3.2-final-0 --------------------------------------------------
Name
         Stmts   Miss  Cover
--------------------------------------------------------------------------------------
----------------------------
control_structures_and_functions/a_if_statement/if_statement_multiple_lines
            11      0   100%
control_structures_and_functions/a_if_statement/if_statement_repeating_variable_name
            10      0   100%
control_structures_and_functions/a_if_statement/make_use_of_pythons_truthiness
            20      3    85%
control_structures_and_functions/b_for_loop/enumerate
            10      0   100%
control_structures_and_functions/b_for_loop/in_statement
            10      0   100%
```

```
control_structures_and_functions/b_for_loop/use_else_to_determine_when_break_not_hit
            31      0   100%
control_structures_and_functions/functions/2only/2only_use_print_as_function
             4      0   100%
control_structures_and_functions/functions/avoid_list_dict_as_default_value
            22      0   100%
control_structures_and_functions/functions/use_args_and_kwargs_to_accept_arbitrary_arg
uments      39     31    21%
control_structures_and_functions/zexceptions/aaa_dont_fear_exceptions
             0      0   100%
control_structures_and_functions/zexceptions/aab_eafp
            22      2    91%
control_structures_and_functions/zexceptions/avoid_swallowing_exceptions
            17     12    29%
general_advice/dont_reinvent_the_wheel/pypi
             0      0   100%
general_advice/dont_reinvent_the_wheel/standard_library
             0      0   100%
general_advice/modules_of_note/itertools
             0      0   100%
general_advice/modules_of_note/working_with_file_paths
            39      1    97%
general_advice/testing/choose_a_testing_tool
             0      0   100%
general_advice/testing/separate_tests_from_code
             0      0   100%
general_advice/testing/unit_test_your_code
             1      0   100%
organizing_your_code/aa_formatting/constants
            16      0   100%
organizing_your_code/aa_formatting/formatting
             0      0   100%
organizing_your_code/aa_formatting/multiple_statements_single_line
            17      0   100%
organizing_your_code/documentation/follow_pep257
             6      2    67%
organizing_your_code/documentation/use_inline_documentation_sparingly
            13      1    92%
organizing_your_code/documentation/what_not_how
            24      0   100%
organizing_your_code/imports/arrange_imports_in_a_standard_order
             4      0   100%
organizing_your_code/imports/avoid_relative_imports
             4      0   100%
organizing_your_code/imports/do_not_import_from_asterisk
             4      0   100%
organizing_your_code/modules_and_packages/use_modules_where_other_languages_use_object
             0      0   100%
organizing_your_code/scripts/if_name
            22      0   100%
organizing_your_code/scripts/return_with_sys_exit
            32      2    94%
working_with_data/aa_variables/temporary_variables
            12      0   100%
working_with_data/ab_strings/chain_string_functions
```

```
            10      0   100%
working_with_data/ab_strings/string_join
            10      0   100%
working_with_data/ab_strings/use_format_function
            18      0   100%
working_with_data/b_lists/2only/2only_prefer_xrange_to_range
            14     14     0%
working_with_data/b_lists/3only/3only_unpacking_rest
            16      0   100%
working_with_data/b_lists/list_comprehensions
            13      0   100%
working_with_data/ca_dictionaries/dict_dispatch
            23      0   100%
working_with_data/ca_dictionaries/dict_get_default
            10      1    90%
working_with_data/ca_dictionaries/dictionary_comprehensions
            21      0   100%
working_with_data/cb_sets/make_use_of_mathematical_set_operations
            25      0   100%
working_with_data/cb_sets/set_comprehensions
            12      0   100%
working_with_data/cb_sets/use_sets_to_remove_duplicates
            34      6    82%
working_with_data/cc_tuples/named_tuples
            26      0   100%
working_with_data/cc_tuples/tuple_underscore
            15      0   100%
working_with_data/cc_tuples/tuples
            12      0   100%
working_with_data/classes/2only/2only_prepend_private_data_with_underscore
            43     43     0%
working_with_data/classes/2only/2only_use_str_for_human_readable_class_representation
            18     18     0%
working_with_data/classes/3only/3only_prepend_private_data_with_underscore
            45      2    96%
working_with_data/classes/3only/3only_use_str_for_human_readable_class_representation
            18      0   100%
working_with_data/context_managers/context_managers
            16      7    56%
working_with_data/generators/use_generator_expression_for_iteration
            16      0   100%
working_with_data/generators/use_generators_to_lazily_load_sequences
            44      1    98%
--------------------------------------------------------------------------------------
----------------------------
TOTAL
           849    146    83%

============================================================ 150 passed in 1.73 second
s ============================================================
_____________________________________________________________________ summary ________
______________________________________________________________
py33verbose: commands succeeded
congratulations :)
```

That's pretty damn awesome.

`setuptools` integration

`tox` can be integrated with `setuptools` so that `python setup.py test` runs your `tox` tests. The following snippet should be put in your `setup.py` file and is taken directly from the `tox` documentation:

```
from setuptools.command.test import test as TestCommand
import sys

class Tox(TestCommand):
    def finalize_options(self):
        TestCommand.finalize_options(self)
        self.test_args = []
        self.test_suite = True
    def run_tests(self):
        #import here, cause outside the eggs aren't loaded
        import tox
        errcode = tox.cmdline(self.test_args)
        sys.exit(errcode)

setup(
    #...,
    tests_require=['tox'],
    cmdclass = {'test': Tox},
    )
```

Now `python setup.py test` will download `tox` and run `tox`. Seriously cool. And a serious time saver.

Documentation with *Sphinx*

Sphinx (http://www.sphinx-doc.org) is a tool by the pocoo (http://www.pocoo.org/) folks. It's used to generate the Python's official documentation and the documentation for almost all other popular Python packages. It was written with idea of making auto-generation of HTML documentation from Python code as easy as possible.

Let the tool do the work

Sphinx has no implicit knowledge of Python programs and how to extract documentation from them. It can only translate reStructured Text files, which means a reStructured Text version of your code's documentation needs to be available for Sphinx to do its work. But maintaining a reStructured Text version of all of your `.py` files (minus the actual body of functions and classes) is clearly not doable.

Luckily, Sphinx has a javadoc-like extension, called `autodoc`, which is able to extracted reStructured Text from your code's docstrings. To be able to fully utilize the power of Sphinx and `autodoc`, you'll need to format your docstrings in a particular manner. In particular, you should make use of Sphinx's Python directives. Here's an example of a function documented using reStructured Text directives, making the resulting HTML documentation much nicer:

```
def _validate(cls, method, resource=None):
"""Return ``True`` if the given *cls* supports the HTTP *method* found
on the incoming HTTP request.

:param cls: class associated with the request's endpoint
:type cls: :class:`sandman.model.Model` instance
:param string method: HTTP method of incoming request
:param resource: *cls* instance associated with the request
:type resource: :class:`sandman.model.Model` or None
:rtype: bool

"""
if not method in cls.__methods__:
    return False

class_validator_name = 'validate_' + method

if hasattr(cls, class_validator_name):
    class_validator = getattr(cls, class_validator_name)
    return class_validator(resource)

return True
```

Documentation becomes a bit more work, but the payoff is worth it for your users. Good, accessible documentation sets a usable project apart from a frustrating one.

Sphinx's `autodoc` extension gives you access to a number of directives that automatically generate documentation from your docstrings.

Installation

Be sure to install Sphinx *in your* `virtualenv` , since documentation will be a versioned artifact in your project. Different versions of Sphinx may generate different HTML output. By installing in your `virtualenv` , you can "upgrade" your documentation in a controlled manner.

We'll be keeping our documentation in the `docs` directory and the generated documentation in the `docs/generated` directory. To auto-generate reStructured Text documentation files from your `docstring` s, run the following command in your project's root directory:

```
$ sphinx-apidoc -F -o docs <package name>
```

This will create a `docs` directory with a number of documentation files. In addition, it creates a `conf.py` file, which is responsible for configuration of your documentation. You'll also see a `Makefile`, handy for building HTML documentation in one command (`make html`).

Before you actually generate your documentation, be sure you've installed your package locally (`$ python setup.py develop` is the easiest way to keep it up to date, though you can use `pip` as well) or else `sphinx-apidoc` won't be able to find your package.

Configuration: `conf.py`

The `conf.py` file that was created controls many aspects of the documentation that's generated. It's well documented itself, so I'll briefly touch on just two items.

version and release

First, make sure to keep your `version` and `release` values up-to-date. Those numbers will be displayed as part of the generated documentation, so you don't want them to drift from the actual values.

The easiest way to keep your version up to date, in both your documentation and `setup.py` file, is to have it read from your package's `__version__` attribute. I "borrowed" the following `conf.py` code for `sandman` from Flask's `conf.py`:

```
import pkg_resources
try:
    release = pkg_resources.get_distribution('sandman').version
except pkg_resources.DistributionNotFound:
    print 'To build the documentation, The distribution information of sandman'
    print 'Has to be available.  Either install the package into your'
    print 'development environment or run "setup.py develop" to setup the'
    print 'metadata.  A virtualenv is recommended!'
    sys.exit(1)
del pkg_resources

version = '.'.join(release.split('.')[:2])
```

This means that, to get the documentation to generate the correct version number, you simply need to have run `$ python setup.py develop` in your project's `virtualenv`. Now you only need to worry about keeping `__version__` up to date, since `setup.py` makes use of it as well.

html_theme

Consider changing the `html_theme` from `default`. I'm partial to `nature`, obviously this is a matter of personal preference. The reason I raise this point at all is because the official Python documentation changed themes from `default` to `pydoctheme` between Python 2 and Python 3 (the latter theme is a custom theme only available in the cPython source). To some people, seeing the `default` theme makes a project seem "old".

PyPI

PyPI, the Python Package Index (http://pypi.python.org/pypi) (formerly known as "the Cheeseshop") is a central database of publicly available Python packages. PyPI is where your project's releases "live." Once your package (and its associate meta-data) has been uploaded to PyPI, others can download and install it using `pip` or `easy_install`. This point bears repeating: *even if your project is available on GitHub, it's not until a release is uploaded to PyPI that your project is useful*. Sure, someone *could* clone your git repo and manually install it directly, but *far* more people just want to `pip install` it.

One last step

If you've completed all of the steps in the previous sections, you're likely anxious to bundle up your package, upload it to PyPI, and make it available to the world!

Before you do so, however, there's a helpful tool called `cheesecake` that is helpful to run as the last step before distributing your package. It analyzes your package and assigns "scores" in a number of categories. It measures how easy/correct packaging and installing your package is, the quality of the code, and the quality and quantity of your documentation.

As a coarse measure of "readiness", `cheesecake` is great for sanity checking. You'll quickly see if there's an issue with your `setup.py` file or if you forgot to document a file. I recommend running it before *each* upload to PyPI, not just the first one.

Initial upload

Now that you've confirmed your code isn't crap and won't break when people try to install it, let's get your package on PyPI! You'll be interacting with PyPI through `setuptools` and the `setup.py` script. If this is the first time this particular package is being uploaded to PyPI, you'll first need to *register* it:

```
$ python setup.py register
```

Note: if you don't yet have a free PyPI account, you'll need to make one now to be able to register the package. After you've followed `register` prompts, you're ready to create your distributable package and upload it to PyPI:

```
$ python setup.py sdist upload
```

The command above builds a source distribution (`sdist`) and uploads it to PyPI. If your package isn't pure Python (that is, you have binaries that need to be built), you'll need to do a binary distribution. See the `setuptools` documentation for more info.

Releases and version numbers

PyPI uses a *release version* model to decide which version of your package should be available by default. After the initial upload, you'll need to create a *release* with a new *version number* each time you want your updated package to be made available on PyPI. Managing your version number can actually be a fairly complex topic, so much so that there's a PEP for it: PEP 440 -- Version Identification and Dependency Specification (http://www.python.org/dev/peps/pep-0440/). I'd definitely suggest following the guidelines in PEP 440 (obviously), but if you choose to use a different versioning scheme, the `version` used in `setup.py` **must** be "higher" than what's currently on PyPI for PyPI to consider the package a new version.

Workflow

After uploading your first release to PyPI, the basic workflow is this:

1. Do some work on your package (i.e. fix bugs, add features, etc)
2. Make sure the tests pass
3. "Freeze" your code by creating a `release` branch in git-flow
4. Update the `__version__` number in your package's `__init__.py` file
5. Run `python setup.py sdist upload` to upload the new version of your package to PyPI

Users depend on you to release frequently enough to get bug fixes out. As long as you're properly managing your version numbers, there is no such thing as releasing "too frequently." Remember: your users aren't manually maintaining the different versions of every Python package they have installed.

Continuous Integration with TravisCI

Continuous Integration refers to the process of continuously integrating all changes for a project (rather than periodic bulk updates). For our purposes, it means that *each time we push a commit to GitHub our tests run, telling us if the commit broke something.* As you can imagine, this is an incredibly valuable practice. There's no more "forgetting to run the tests" before committing/pushing. If you push a commit that breaks the tests, you'll get an email telling you so.

TravisCI (http://www.travis-ci.org) is a service that makes continuous integration for GitHub projects embarrassingly easy. Head over there and create an account if you don't yet have one. Once you're done, we'll need to create one simple file before we're swimming in CI goodness.

Configuration via `.travis.yml`

Individual projects on TravisCI are configured via a file, `.travis.yml`, in the project's root directory. Briefly, we need to tell Travis:

1. What language our project is written in

2. What version of that language it uses
3. What commands are used to install it
4. What commands are used to run the project's tests

Doing so is quite straightforward. Here are the contents of the `.travis.yml` file from sandman (http://www.github.com/jeffknupp/sandman):

```
language: python
python:
    - "2.7"
install:
    - "pip install -r requirements.txt --use-mirrors"
    - "pip install coverage"
    - "pip install coveralls"
script:
    - "coverage run --source=sandman setup.py test"
after_success:
    coveralls
```

After listing the language and version, we tell Travis how to install our package. Under `install:` , make sure you have the line:

```
- "pip install -r requirements.txt --use-mirrors"
```

This `pip install`s our projects requirements (and uses PyPI mirrors if necessary). The other two lines in `install` are specific to sandman (http://www.github.com/jeffknupp/sandman). It's using an additional service (coveralls.io (http://coveralls.io)) to continuously monitor test case coverage, but that's not necessary for all projects.

`script:` lists the command needed to run the project's tests. Again, sandman (http://www.github.com/jeffknupp/sandman) is doing some extra stuff. All your project needs is `python setup.py test` . And the `after_success` portion can be dropped all together.

Once you've committed this file and activated your project's repo in TravisCI, push to GitHub. In a few moments, you should see a build kick off on TravisCI based on your most recent commit. If all is successful, you build will be "green" and the status page will show that the build passed. You'll be able to see the history of all of your project's builds at any time. This is especially useful for multi-developer projects, where the history page can be used to see how often a particular developer breaks the build...

You should also receive an email letting you know the build was successful. Though you can probably configure it otherwise, you'll get emails only when the build is broken or fixed, but not if a commit has the same outcome as the build that preceded it. This is incredibly useful, as your not inundated by useless "the build passed!" emails but are still alerted when something changes.

ReadTheDocs for Continuous Documentation Integration

While PyPI has an official documentation site (pythonhosted.org (http://www.pythonhosted.org)), ReadTheDocs (https://readthedocs.org/) provides a better experience. Why? ReadTheDocs has great integration with GitHub. Once you register on ReadTheDocs, you'll see all of your GitHub repos. Select the appropriate repo, do some minor configuration, and your documentation will be automatically regenerated after each commit to GitHub.

Configuring your project should be a straightforward affair. There are a few things to remember, though. Here's a list of configuration fields and the values you should use which might not be immediately obvious:

- Repo: https://github.com/*github_username*/*project_name*.git
- Default Branch: `develop`
- Default Version: `latest`
- Python configuration file: (leave blank)
- Use `virtualenv`: (checked)
- Requirements file: `requirements.txt`
- Documentation Type: Sphinx HTML

Don't Repeat Yourself

Now that you've done all that hard work to open-source an existing code base, you likely don't want to have to repeat it all when starting a *new* project. Luckily, you don't have to. Audrey Roy's Cookiecutter (https://github.com/audreyr/cookiecutter-pypackage) tool (I've linked to the Python version, though there are versions for numerous languages in the main repo (https://github.com/audreyr/cookiecutter)).

Cookiecutter is a command line tool that automates the process of starting a project in a way that makes doing the stuff discussed in this article easy. Daniel Greenfeld (@pydanny (http://www.twitter.com/pydanny)) wrote a great blog post about it and how it relates to the practices discussed in this article. You should check it out: Cookiecutter: Project Templates Made Easy (http://pydanny.com/cookie-project-templates-made-easy.html).

Conclusion

We've now covered all of the commands, tools, and services that go into open sourcing an existing Python package. Sure, you could have just thrown it on GitHub and said, "install it yourself," *but no one would have.* And you wouldn't *really* have Open Source Software; you'd simply have "free code."

What's more, you likely never would have attracted outside contributors to your project. By setting up your project in the manner outlined here, you've created an easy to maintain Python package that encourages *both use and contribution*. And that, after all, is the true spirit of Open Source Software, is it not?

Posted on Aug 16, 2013 by Jeff Knupp

> Discuss Posts With Other Readers at **discourse.jeffknupp.com (http://discourse.jeffknupp.com)**!

« My Favorite Creation (/blog/2013/08/09/my-favorite-creation)

Like this article?

Why not sign up for **Python Tutoring**? Sessions can be held remotely using Google+/Skype or in-person if you're in the NYC area. Email jeff@jeffknupp.com (mailto:jeff@jeffknupp.com) if interested.

Sign up for the free jeffknupp.com email newsletter. Sent roughly once a month, it focuses on Python programming, scalable web development, and growing your freelance consultancy. And of course, you'll never be spammed, your privacy is protected, and you can opt out at any time.

Email Address

Subscribe

42 Comments jeffknupp.com Login

Sort by Best Share Favorite ★

Join the discussion…

audreyr · a year ago

I found out recently that importing the package into setup.py to get package.__version__ can cause all sorts of problems. See https://github.com/audreyr/coo... for more info.

Btw, thanks for the Cookiecutter mention!

6 ⌃ | ⌄ • Reply • Share ›

moliware ➦ audreyr • a year ago

+1, I had problems with that too.

⌃ | ⌄ • Reply • Share ›

Paul Winkler ➦ audreyr • a year ago

True. "All sorts of problems" == your package cannot be installed unless everything it imports is already pre-installed.

https://github.com/jeffknupp/s...

⌃ | ⌄ • Reply • Share ›

jeffknupp Jeff ➦ Paul Winkler • a year ago

Yeah, just fixed this and closed the issue. Thanks for the heads up. I had actually already written an alternate way of getting the version in the `find_version` function but forgot to actually use it.

⌃ | ⌄ • Reply • Share ›

Jan-Philip Gehrcke ➦ jeffknupp • 6 months ago

But you still did not update the code in the article above, right? :) There still is a `import sandman` in the `setup.py`.

In one of my projects I was also bitten by this issue, see https://bitbucket.org/jgehrcke....

⌃ | ⌄ • Reply • Share ›

numerodix • a year ago

- That, I believe, is not a point to be taken likely.

lightly

2 ⌃ | ⌄ • Reply • Share ›

jeffknupp Jeff ➦ numerodix • a year ago

Good catch! Fixed.

⌃ | ⌄ • Reply • Share ›

Daniel Greenfeld • a year ago

What's awesome is that just recently Audrey Roy created a project template for her cookiecutter tool that implements nearly everything Jeff talks about. Which means following his practices is really easy. You can find it here:
https://github.com/audreyr/coo...
https://github.com/audreyr/coo...

2 ⌃ | ⌄ • Reply • Share ›

jeffknupp Jeff ➦ Daniel Greenfeld • a year ago

Updated to include the info!

2 ⌃ | ⌄ • Reply • Share ›

Geza • 4 months ago

Great article, thank you! A suggestion: If you include a menu, with #links to sections of the document, it would make referencing parts of the tutorial easier. Thanks!

1 ⌃ | ⌄ • Reply • Share ›

Bernardo Brik • a year ago

Hi, great article, very helpful!
What if my project has internationalization? I know I can mark the strings as translatable, but how do I set which is the app language?

In full django projects you set LANGUAGE_CODE, but how to do it in packages?
Thanks!

1 ⌃ | ⌄ • Reply • Share ›

erikb85 • a year ago

I disagree with too many of your points to list them all in form of comments. I think your guide would improve a lot, if you could read up what other (preferably known, influential) people have to say about those topics and reference them. Doing this you will either see, that many people disagree with what you have written so far or you will feel everything gets validated and a reader like me might be able to learn something new. :-)

In any case kudos for all the work, that you've put into that guide!

1 ⌃ | ⌄ • Reply • Share ›

jeffknupp Jeff ➦ erikb85 • a year ago

Can you give an example of anything you mention in your comment? At least share *one thing* you disagree with. Also, who are the "known, influential" people that I should be talking to about this? A number of the Python core devs have responded positively to this article and, if improvements were suggested, I've updated this guide to include them.

I didn't want to mark this comment as spam as I genuinely want to improve this guide where possible, but as it stands your comment adds nothing to the discourse on this topic and is toeing the "spam" line...

12 ⌃ | ⌄ • Reply • Share ›

sly • a month ago

Great article, thank you.

One little typo has still been overseen: "Actually running your all of your tests against all of your environments now takes four keystrokes:" -> a "your" too much.

⌃ | ⌄ • Reply • Share ›

Olivia Jennifer · 2 months ago

Yeah its a
good article. According to you what we project managers do is communicating.
And a lot of this communication is done during project meetings. It can
sometimes feel like you are running from one meeting to another and that your
time is often wasted. Meetings don’t start on time, the issues aren’t dealt
with, there is no agenda, there is no focus, nobody assigns any follow ups or
tasks and of course then they also don’t end on time. An efficient project manager is required
for the good management of a project. I think a project manager should
PMP certified. Looking forwards to apply what I learned in PMP classes in
my company.

⌃ | ⌄ • Reply • Share ›

jeffknupp Jeff ➦ Olivia Jennifer · 2 months ago

Spam.

1 ⌃ | ⌄ • Reply • Share ›

thefinnomenon · 5 months ago

Great article! I am currently in the process of getting my development environment setup & had pieced together some of these solutions but to get a complete overview is much appreciated.

⌃ | ⌄ • Reply • Share ›

Jan-Philip Gehrcke · 6 months ago

A great article, thanks. It covers many really important topics.

Two additions:

1) As long as setuptools sends the PyPI credentials in clear text over the wire, one should at least warn users in as many places as possible, so a comment about this would not hurt here. Another thing you
could mention is that in the future https://pypi.python.org/pypi/t... will be the standard tool for uploading projects to PyPI (although it is still in its infants).

2) Regarding the versioning topic: http://semver.org/ is always worth a read and should be the recommended resource, especially for libraries.

Thanks, once again!

⌃ | ⌄ • Reply • Share ›

Martin · 6 months ago

Jeff, the markup looks a little messed up right now. Headers don't display correctly. The code snippet with the _validate function is not indented.

Thanks again for the awesome article!

⌃ | ⌄ • Reply • Share ›

Rakuzi · 8 months ago

I have made a simple project using urwid. I want to include urwid in my project so that it don't have to download separately. Can you please tell me how to do this and in which directory have to download separately. Can you please tell me how to do this and in which directory urwid go and how to import it. I tried many times but no success.

^ | v • Reply • Share ›

Alexey Balmashnov • 10 months ago

Nice overview... Have a question though: is it possible to organize similar setup in "closed" (aka corporate) environment preferably employing open source tools?

Would be really nice if you could elaborate on this.

^ | v • Reply • Share ›

jeffknupp Jeff ➦ Alexey Balmashnov • 10 months ago

It's largely the same process, with corporate versions of all of the services mentioned (Travis CI, GitHub, etc). So it really depends on the tools you have access to at your company. At my company, for example, I have a setup quite similar to this but using Jenkins CI for continuous integration. It's important to stay flexible and remember the goal is to have a project that's automatically tested and documented upon commit. How you get there is up to you.

^ | v • Reply • Share ›

Alexey Balmashnov ➦ jeffknupp • 10 months ago

Thank you for your answer. Do I understand it correctly that all of mentioned tools do have their "corporate" flavors which can be used to organize the same process internally on the servers of the company or one still would need to hunt for some substitutions here and there to achieve setup with similar set of features?

^ | v • Reply • Share ›

gwideman • a year ago

"I'd definitely suggest following the guidelines in PEP 400 (obviously)" Probably you mean to repeat PEP440 mentioned in the previous sentence.

^ | v • Reply • Share ›

JBW ➦ gwideman • 10 months ago

"Select the appropriate repo, do some minor configuration, and you're documentation will be automatically regenerated after each commit to GitHub." -- *your

^ | v • Reply • Share ›

jeffknupp Jeff ➦ JBW • 10 months ago

Both fixed. *Sigh*. This is what happens when you don't have an editor...

^ | v • Reply • Share ›

Michael Dunn • a year ago

Jeff, great post. I think there is a typo in the first paragraph under 'setuptools and the setup.py File'. I believe that you mean for the phrase 'It allows packages to be searched for an installed in a programmatic way' to be 'It allows packages to be searched for AND installed in a programmatic way'

Thanks for sharing with the community.

⌃ | ⌄ • Reply • Share ›

Jelle Smet • a year ago

What kind of bothered my though is that the git flow extension seems to be abandoned: https://github.com/nvie/gitflo...

⌃ | ⌄ • Reply • Share ›

> **jeffknupp** Jeff ➦ Jelle Smet • 10 months ago
>
> It honestly doesn't matter if it's maintained or not, since it is feature complete and the branching model itself isn't likely to change.
>
> ⌃ | ⌄ • Reply • Share ›

Jeremy T • a year ago

This is fantastic. Are there any similar guides out there for other languages, particularly javascript?

⌃ | ⌄ • Reply • Share ›

> **jeffknupp** Jeff ➦ Jeremy T • a year ago
>
> Not sure about an article like this, but **@audreyr** has a great project, cookiecutter, which has a javascript plugin: https://github.com/audreyr/coo...
>
> ⌃ | ⌄ • Reply • Share ›

Nathan Goldbaum • a year ago

I'm involved with a pretty active open source project that uses mercurial and is hosted on bitbucket. For most real-world usages github and bitbucket are at feature parity and ditto for mercurial and git. It's a bit of a learning curve and I guess we may scare away new contributors since they have to learn a new VCS, but despite that it seems to be working pretty well for us so far.

⌃ | ⌄ • Reply • Share ›

> **Xiong Chiamiov** ➦ Nathan Goldbaum • a year ago
>
> It's not just learning a new VCS (and the tools that go along with it); it's a matter of not using the same project management site.
>
> All of my code is on Github, work and personal. I check my notifications every day (as part of work), and thus see notifications for any of the numerous projects I've got notifications for (I created an issue, commented on an issue, or just signed myself up for news on one). I've got a news feed of things interesting people are working on, and frequently add people to that list when I see a cool project. If it's code, it's on Github.
>
> So, asking me to do anything with a project on another hosting site (other than downloading it) is like asking someone who has their whole life on Facebook to chime in on a thread on Google Plus. It's not impossible, it's just irritating.
>
> There are legitimate reasons for not using Github (Mercurial tends to have a more sane UX design, Gitorious is open-source), but you have to realize that one of the reasons *for* using Github is that most people are already there - and that's an important factor

now that we're doing "social coding".

1 ⌃ | ⌄ • Reply • Share ›

jeffknupp Jeff ➦ Nathan Goldbaum • a year ago

As I said, I'm not saying it's "right", but git and GitHub is likely more familiar to potential contributors than mercurial and Bitbucket. If one goal of your project is to encourage contribution (and to some extent, it should be, or else it's more of a personal project you happen to be make the code available for), GitHub is a much better choice. Feature-wise, there's almost no difference (although *many* more third-party sites integrate with GitHub than Bitbucket).

⌃ | ⌄ • Reply • Share ›

cjrh • a year ago

"each time we push a commit to GitHub, our tests our run to tell us if the commit broke something."

Something is funny in the region of "our tests our".

⌃ | ⌄ • Reply • Share ›

ram ➦ cjrh • a year ago

Should be "our tests *are* run".

⌃ | ⌄ • Reply • Share ›

jeffknupp Jeff ➦ ram • a year ago

Fixed! Thanks for the heads up.

⌃ | ⌄ • Reply • Share ›

Simone Soldateschi • a year ago

Thanks for sharing an article so dense with information, ideas and resources!

~Simone

⌃ | ⌄ • Reply • Share ›

Crad • a year ago

Great writeup... Why not use tests_require instead of putting your test runner in extras_require in your setup.py? From http://pythonhosted.org/distri...

"If your project's tests need one or more additional packages besides those needed to install it, you can use this option to specify them. It should be a string or list of strings specifying what other distributions need to be present for the package's tests to run. When you run the test command, setuptools will attempt to obtain these (even going so far as to download them using EasyInstall). Note that these required projects will not be installed on the system where the tests are run, but only downloaded to the project's setup directory if they're not already installed locally."

Edited to clarify the question on tests_require vs extras_require.

⌃ | ⌄ • Reply • Share ›

jeffknupp Jeff ➦ Crad • a year ago

You're right. It should be in `tests_require` rather than `extras_require`. I'll update the post. Thanks for the tip!

˄ | ˅ • Reply • Share ›