Here-Be-Pythons! Documentation

Maciej Wrześniewski

Aug 09, 2019

TLDR

Here-Be-Pythons!™ holds opinionated best practices & ideas to help you build your next Python project.

The goal is to gather the sweetest & most impactful nuggets of wisdom relating to Python ventures and make them accessible from a single place. Because a little investment in automation and knowledge makes your snake-projects fly, following the famous equation: + =

Hopefully, Here-Be-Pythons! will inspire you to code your awesome projects faster and to make them even better.

Ahoy, Captain! Thar's opinions ahead!

The content of this repo is based on the research, experience, but also *pure opinions* of the authors. And opinions do differ!

Still, we are more than happy to hear your feedback. All for the glory of this repo and to constantly grow as programmers

---

CHAPTER 1

Table of Contents

1.1 TL;DR Checklist

- Python:
  - Use Python 3.7.
  - Manage app dependencies using a combo of virtualenv and pip-tools.
  - Sort imports with isort.
  - Format source files with YAPF.
  - Lint code with Flake8.
  - Check security issues with Bandit.
  - Use pytest as test framework.
- Git:
  - Use .gitignore.
  - Use Pre-commit Git Hook.
- Project:
  - Store config (credentials, secrets, etc.) in the environment.
  - Manage and execute command line tasks via Invoke.
  - Log, log, log.
  - Maintain up-to-date README.
  - Set up Continuous Integration.

1.2 Big-Bang-py

To start your new Python project, it is recommended to use Big-Bang-py Cookiecutter, as it follows the ideas and principles of Here-Be-Pythons!

Moreover, it is maintained by the very same authors , so both projects are promised to be kept in sync .

1.2.1 Usage

```
# Generate a new project.
> cookiecutter gh:CapedHero/big-bang-py

# Answer all of the prompted questions.
# Brackets show default options. Click <enter> if you wish to accept them.
project_name [My New Project]: ???
project_dir  [my-new-project]: ???
project_source_code_dir [src]: ???

# Finish with:
> cd $MY_NEW_PROJECT_DIR
> ./finish_project_setup
> ./invoke_bash_completion \
      && rm -f invoke_bash_completion \
      && source $VIRTUAL_ENV/bin/activate
```

And voilà! You are ready to code!

1.3 Interpreter Version

Main point

It is recommended to use Python 3.7.

- There is an informative summary of new features on Real Python. More detailed description can be found in the official documentation.
- To name some notable additions:
  - Data Classes can easily replace namedtuples and attrs.
  - Typing Forward Reference makes type hints even more programmer-friendly.
  - The "asyncio" module has received many new features along with usability and performance improvements. For instance, new "asyncio.run()" automatically creates an event loop, runs a task on it and finally closes the loop when the task is done.
- Nonetheless, if you still use previous version of Python (especially 3.6), it is possible to backport some of the new features e.g. Data Classes.

1.4 Dependencies Management

Main point

Use a combo of virtualenv and pip-tools to manage Python app dependencies.

There are two general approaches to Python app dependencies management:

1. Use a classic combination of virtualenv + requirements.
2. Use some "fancy" new tool like Pipenv or Poetry.

If we go with option number 1, we will miss some truly powerful features, such as:

- clean & simple approach to deterministic, hash-based builds;
- easy management of multi-tier dependencies (like `base`, `dev`, `tests`, `prod` split);
- clear separation of main dependencies from sub-dependecies.

On the other hand, new alternatives are usually very bloated, slow and extremely opinionated.

That is why a fusion of virtualenv and pip-tools is so powerful. It seems to offer the best of two worlds:

- It is fast.
- It is based on good, old virtualenv.
- It is based on good, old requirements... with a twist! We put the actual dependencies in `requirements.in` and pip-tools compiles actual dependencies with all sub-dependencies into their locked, hashed versions in `requirements.txt`.

  In result, we have simple to manage, hash-based, deterministic builds. Nice!

So give it a try. The sooner you start, the sooner you will fall in love with it

1.4.1 Proposed workflow

```
> cd $PYTHON_PROJECT_DIR

# Below approach seems to be the simplest and the most effective way to handle␣
↪virtualenvs.
# 1. It doesn't clutter your filesystem.
# 2. You always know the location and the name of the environment.
# 3. PyCharm happily finds the virtualenv as soon as you open the project.
# 4. It just works. No need for virtualenvwrappers, burritos, Bash aliases, etc.
> virtualenv venv

> source venv/bin/activate

# Don't you dare using pip-tools from the root Python environment!
# Always use the version from the virtualenv that is locked in your main dependencies␣
↪file.
# See: https://github.com/jazzband/pip-tools/issues/328#issuecomment-346685547
> pip install pip-tools

# Assumption for the next steps:
# You have already prepared dependencies files base.in and dev.in in ./requirements/
↪abstract
> pip-compile \
    --generate-hashes \
```

(continues on next page)

(continued from previous page)

```
    --output-file \
    requirements/locked/base.txt \
    requirements/abstract/base.in
> pip-compile \
    --generate-hashes \
    --output-file \
    requirements/locked/dev.txt \
    requirements/abstract/dev.in
> pip-sync requirements/locked/dev.txt
# You can handle the above shell commands via Invoke tasks.
# See an example: https://git.io/fjcZ6
```

- Big-Bang-py is based on the above workflow.
- Remember that both abstract and locked dependencies should be checked in the Git repository.
- If you want to have a deterministic build (e.g. in Docker container), Python app dependencies are installed as simple as:

```
pip install -r requirements/locked/*YOUR_REQUIREMENTS_FILE*.txt
```

1.5 isort - Imports Sorter

Main point

Use isort to automatically sort and group Python imports.

- There are several knobs configuring isort's behavior. Full reference of settings can be found on the isort wiki.
- You can specify project level configuration by placing `.isort.cfg` file at the root of your project.
  - An example of preconfigured `.isort.cfg` is in Big-Bang-py.
- It is recommended to include isort in your linting Invoke task and also to run it during Pre-commit Git Hook. Example of both can be found in Big-Bang-py, see project.py and pre-commit files.
- To manage edge cases, disable isort:
  - per line:

```
from spam import gouda, eggs  # isort:skip

# OR

from foo import (  # isort:skip
    baz,
    bar,
)
```

  - per file:

```
"""
Make the best, yet wrongly sorted, spam & eggs.

NOTE!
```

(continues on next page)

(continued from previous page)

```
To make isort skip an entire file simply, add `isort:skip_file` somewhere
in module's docstring, just like below:

isort:skip_file
"""

import spam
import eggs
```

1.6 Black - The Uncompromising Formatter

Main point

Black's goal is to make Python code look as good as written by a programmer who is rigorously following a style guide. It proved to be much better than any of its alternatives.

Make it your favorite formatter right now.

- Black reformats entire files in place. **It is not configurable!** Thanks to that:
  - Blackened code will look the same regardless of the project you're reading. Formatting becomes transparent after a while and you can focus on the content instead.
  - **Black makes code review much, much faster!** It's been proven empirically on every single project it was applied.
  - All in all, you will save time and mental energy for more important matters than formatting, while keeping your code straight, beautiful and readable.
- It is recommended to include Black in your linting Invoke task and also to run it during pre-commit Git Hook. Example of both can be found in Big-Bang-py, see project.py and CI checks called by Git Hook.
- To manage edge cases, disable Black per block:

```
# fmt: off

all_domains_dict_1 = {
    'valid-safe-foo-1.bar': {'impression': 1},
    'valid-safe-foo-2.bar': {'impression': 2},
}

all_domains_dict_2 = {
    'valid-safe-spam-3.eggs': {'impression': 3},
    'valid-safe-spam-4.eggs': {'impression': 4},
}
unsafe_domains_dict_2 = {
    'valid-unsafe-spam-1.eggs': {'impression': 5},
    'valid-unsafe-spam-2.eggs': {'impression': 6},
}

# fmt: on
```

1.6.1 YAPF - Google's Alternative

- The formatting style used by YAPF is configurable, where specific configuration can be pointed in a couple of ways.
  - It is recommended to store the configuration in a properly formatted `.style.yapf` file at the root of your project.
  - You may find pre-configured `.style.yapf` in Big-Bang-py.
- It is recommended to include YAPF in your linting Invoke task and also to run it during pre-commit Git Hook. Example of both can be found in Big-Bang-py, see project.py and pre-commit.
- To manage edge cases, disable YAPF:
  - per line:

```
INTERVALS = (
    ('weeks',   60 * 60 * 24 * 7),
    ('days',    60 * 60 * 24),
    ('hours',   60 * 60),
    ('minutes', 60),
    ('seconds', 1),
)  # yapf: disable
```

  - per block:

```
# yapf: disable

all_domains_dict_1 = {
    'valid-safe-foo-1.bar': {'impression': 1},
    'valid-safe-foo-2.bar': {'impression': 2},
}

all_domains_dict_2 = {
    'valid-safe-spam-3.eggs': {'impression': 3},
    'valid-safe-spam-4.eggs': {'impression': 4},
}
unsafe_domains_dict_2 = {
    'valid-unsafe-spam-1.eggs': {'impression': 5},
    'valid-unsafe-spam-2.eggs': {'impression': 6},
}

# yapf: enable
```

- Survival tips:
  - If you leave trailing comma in a collection (be it a list, function parameters, etc.), YAPF will force it to break, giving one element per line.
  - YAPF is not perfect - from time to time you WILL see weirdly formatted code. There are at least two major occurrences:
    * Deeply Nested Dicts - this is quite understandable as decisions that improve readability are usually arbitrary and should be solved on a case-by-case basis.
    * Complex Comprehensions - comprehensions are split over multiple lines only when they exceed the column limit... This issue is brought to the attention of both YAPF authors (see issue on Github) and other programmers (see posts on Reddit) and Stack Overflow).

1.7 Flake8 - Octo-Ninja Linter

Main point

Flake8 combines so much linter-power under the hood of a single tool! If used correctly, it will make your code not only more consistent, but simply better (and more pythonic).

- Flake8 is a wrapper around three tools:
  1. PyFlakes - checks for Python errors.
  2. pycodestyle - tests Python code against some of the style conventions in PEP 8.
  3. McCabe - analyses Python code complexity (see the *next section* for more details).
- Flake8 is configurable, where specific setup can be pointed in a couple of ways.
  - You may find preconfigured .flake8 in Big-Bang-py.
- There is an abundance of plugins greatly extending capability of Flake8. Search for them on GitHub.
  - A bunch of plugins are included in Flake8 configuration of Big-Bang-py. See all *flake8-** packages in dev.in file.
  - An interesting example is flake8-html, which generates readable Flake8 HTML report (works similar to *coverage html*).
- It is recommended to include Flake8 in your linting Invoke task and also to run it during Pre-commit Git Hook. Example of both can be found in Big-Bang-py, see project.py and Pre-commit.
- To manage edge cases, exclude Flake8 checking:
  1. per code line;
  2. per entire file;
  3. per combination of file & error code;
  4. per particular error.

  Option 3 & 4 are best set in Flake8 config. See an example below:

```
ignore =
    # C408: Unnecessary (dict/list/tuple) call - rewrite as a literal
    #
    # Calling directly dict/list/tuple is more obvious & explicit, making␣
↪it
    # easier to read.
    C408,

    # C812 missing trailing comma
    #
    # Experience shows that this can be seriously inconvenient.
    C812

per-file-ignores =
  ci/ci_checks.py: T001
  tasks/release.py: T001
```

1.8 McCabe - Code Complexity Checker

Main point

Library McCabe automatically detects over-complex code basing on cyclomatic complexity.

- Cyclomatic complexity is approximately equivalent to one plus the number of "loops and if statements". The simple interpretation is that it shows an upper bound for the number of test cases required to obtain branch coverage of the code, therefore it roughly indicates the effort required for writing tests.
  - Additional explanations can be found on tutorialspoint and Wikipedia.
- Code with high cyclomatic complexity (usually assumed as 10+) is likely to be difficult to understand and therefore have a higher probability of containing defects.
- It is recommended to include McCabe in your linting Invoke task and also to run it during Pre-commit Git Hook. In Big-Bang-py McCabe is run automatically by Flake8 linter.
  - Cut-off complexity in Invoke task and Pre-commit Git Hook is arbitrarily assumed to be 7 (configured by max-complexity set in .flake8). However, this number should be adjusted to reflect your experience and project needs.

1.9 Bandit - Find Security Issues

Main point

Bandit will report security issues in Python code that might have slipped off your attention.

- Bandit is a tool designed to find common security issues in Python code.
- To run Bandit from the shell, simply call:

```
bandit --recursive .
```

  Details about more advanced usage can be found in Bandit README.
- Bandit is configurable via .bandit file.
  - You may find preconfigured .bandit in Big-Bang-py.
- It is recommended to include Bandit in your linting Invoke task and also to run it during Pre-commit Git Hook & CI. Example of both can be found in Big-Bang-py, see project.py and ci_checks.py.
- To manage edge cases, exclude Bandit checking per line.

1.10 Pytest - Test Framework

Main point

Pytest is a programmer-friendly Python test framework. It makes it easy to write small tests, yet scales to support complex cases.

- Major `pytest` features:
  - Battle-tested and mature.
  - Informative test failures.
  - Less verbose (plain `assert` vs. `self.assertEqual`, `self.assertGreaterEqual`, etc.).
  - Classes are not required.
  - A far more convenient way to write setup & teardown functions with fixtures.
  - Parameterized tests.
  - Fantastic test runner (a.o. marker- and name-based test selection).
  - Rich plugin architecture.
  - Auto-discovery of test modules and functions.
  - Runs `unittest` and `nose` test suites out-of-the-box.
- You can easily set test runner default flags by defining them in a configuration file called `pytest.ini`. You can find an example in Big-Bang-py.
- Recommended test runner plugins:
  - pytest-cov - prints coverage report at the end of test runner report.
  - pytest-mock - adds `mocker` fixture, which makes mocks easier and more readable.
  - pytest-socket - disables socket calls during tests to ensure network calls are prevented. Amazing to protect yourself against incidental DB or API calls.
  - pytest-sugar - makes test runner report even nicer.
- Useful content:
  - Official documentation
  - Python Testing with pytest: Simple, Rapid, Effective, and Scalable

1.11 .gitignore

Main point

ALWAYS use .gitignore. It specifies files intentionally ignored by Git.

- If you are using PyCharm, definitely install .ignore plugin. This will make managing `.gitignore` a breeze.
- Alternatively, it is possible to generate `.gitignore` online using gitignore.io.
- You may find an example of `.gitignore` for Python project in Big-Bang-py.

1.12 Pre-commit Git Hook

Main point

The Pre-commit Git Hook is run - surprise, surprise - before you commit! It is a perfect opportunity to run tests and linters.

- Regardless of your discipline as a software programmer or simply how good is your memory, Pre-commit Git Hook will check your code automatically. In effect:
  - Your code stays consistent and readable.
  - All while keeping to pass both new and regression tests.
- Pre-commit Git Hook is a match made in heaven when combined with properly configured *Continuous Integration*. It can save you a lot of time (and nerves!) when you have a solid opportunity to fail the checks locally i.e. before being officially rejected by your hosted VSC provider.
- You can find an example of Pre-commit Git Hook in Big-Bang-py.
  - `set_precommit` Invoke task in Big-Bang-py takes your Pre-commit Git Hook from `/githooks/pre-commit` and automatically sets it up for you.

1.13 Semantic Commit Messages

Main point

Adding standardized, meaningful prefix to each commit message may significantly improve readability. Also, it facilitates keeping up-to-date CHANGELOG.

- To begin with, let's start with a few examples of semantic commit messages:
  - `feat: Notify shopper about successful product shipment`
  - `tests: Add tests for class <PeriodicPayment>`
  - `rfct: Refactor Invoke tasks for managing Docker containers`
- As you can see, semantic commit messages stand for prefixing commit messages in a standardized, meaningful way. In effect:
  - It is easier to read and scan commit messages (be it during Pull Request, or when you are looking for that-one-particular-bugfix).
  - It makes maintaining CHANGELOG (almost) a pleasure. Simply group commit messages in their respective sections, which means putting all `feat` commits under `Features`, all `rfct` commits under `Refactoring`, etc.
    - An exciting idea is to do it in reverse i.e., first add a new row to CHANGELOG, and then use this row to create the commit message. For instance:

      Under section `Docs` write a new line:

      `- Update 'New Machine Configuration' section in README.`

      Then add above change to CHANGELOG to staging and make a commit with the copied message:

      `docs: Update 'New Machine Configuration' section in README`
- There is no single standard for semantic commit messages prefixes. And this is fine! You should tailor the rules to fit your project's needs. Still, a nice place to start might be the list below:
  - `bugs` = fixes for bugs that are visible for the users

  - `depl` = deployment scripts, Docker configs, etc.
  - `docs` = changes to the documentation
  - `feat` = new feature for the users
  - `front` = frontend changes
  - `misc` = miscellaneous i.e. any other business
  - `perf` = performance improvements
  - `rfct` = refactoring
  - `style` = code formatting
  - `tests` = adding new tests, refactoring existing tests
  - `ver` = new release
- You can educate yourself further by reading how others implement the grand idea of semantic commit messages:
  - joshbuchea's gist
  - Conventional Commits website

1.14 Store config in ENVs

Main point

The Twelve-Factor App stores config in the environment variables.

- This definition of 'config' does not include internal application controls, such as Django or Flask knobs & flags. This type of settings does not vary between deploys, nor contains sensitive credentials, so is best done in the code.
- **A litmus test for whether an app has its config correctly factored out of the code is whether the codebase could be made open source at any moment without compromising any secrets or credentials.**
- Usually there are multiple ENV files e.g. a separate version for development, staging and production.
  - It is convenient to organise those ENVs in one location. An example of such organisation is present in Big-Bang-py. As a bonus, you can find there Python ENVs loader based on python-dotenv.

1.15 Invoke - Manage & Execute Tasks

Main point

Turn into a task every project related shell command which will be called more than a couple of times and is not super-common (like `ls` with basic flags).

Manage and execute those project tasks via Invoke.

- You can replace `Makefiles` and similar task managers straightaway as Invoke is intuitive and user-friendly.
- Invoke tasks are called by typing in the shell `invoke *task-name*`.

  - Invoke can be easily buffed with shell tab completion.

    If you work on your projects using `bash` with virtualenv, a ready-2-go installation script can be found in Big-Bang-py. If your development environment differs, this script can still give you a basis, or at least a hint, how to build a solution of your own.

- Invoke tasks are normal Python functions organised in `tasks.py` file or tasks Python package.
  - You may find examples of Invoke tasks in Big-Bang-py.
- Docstrings of functions implementing Invoke tasks are automatically formatted into a command line help:

```
> invoke --list
Available tasks:

  task1   First line of task1 docstring.
  task2   First line of task2 docstring.

> invoke --help task2
Usage: inv[oke] [--core-opts] task2 [--options] [other tasks here ...]

Docstring:
  Full docstring of task 2.

Options:
  -p TYPE, --param=TYPE   Your additional parameters help string.
```

Note: To learn how to add help for additional parameters for Invoke tasks, see the docs.

- Invoke tasks can be organised using namespaces. For instance:

```
# File: $PROJECT_ROOT/tasks.py

import invoke

import src.app.tasks
import src.db.tasks


@invoke.task
def core_task_1(c):
    pass


@invoke.task
def core_task_2(c):
    pass


###################################
# Organise tasks into namespaces #
###################################

# Because we are customizing tasks, now we HAVE TO manually create
# main namespace and point to it via variable named `namespace` or `ns`.
# See: http://docs.pyinvoke.org/en/1.2/concepts/namespaces.html#starting-
↪out
```

(continues on next page)

(continued from previous page)

```
namespace = invoke.Collection()

# Add to the main namespace top-level tasks from the current file.
namespace.add_task(core_task_1)
namespace.add_task(core_task_2)

# Create `app` namespace and add related tasks.
app_namespace = invoke.Collection('app')
app_namespace.add_task(src.app.tasks.start)
app_namespace.add_task(src.app.tasks.stop)

# Create `db` namespace and add related tasks.
db_namespace = invoke.Collection('db')
# By default task name will be derived from the implementing function
# name, but we can also customize it via `name` argument.
db_namespace.add_task(src.db.tasks.fire_up_postgres, name='fire_up')
db_namespace.add_task(src.db.tasks.stop_postgres, name='stop')
# We can nest `db` namespace into `app` namespace!
app_namespace.add_collection(db_namespace)

# Finally, we have to add `app` namespace (together with the nested
# `db` tasks) to the main namespace.
namespace.add_collection(app_namespace)
```

Now we can call our tasks like `app.start` or `app.db.fire-up`. Sweet!

- If Invoke task behaves weirdly regarding prints/logs/stdout/stderr/etc. it is worth trying to add `pty=True` argument in `c.run` call:

```
@invoke.task
def flake8(c):
    c.run('python -m flake8', pty=True)
```

By default, `run` connects directly to the invoked process and reads its stdout/stderr streams. Some programs will behave differently in this situation compared to using an actual terminal or pseudoterminal (pty). Due to their nature, ptys have a single output stream, so the ability to tell stdout apart from stderr is not possible. As such, all output will appear on `out_stream` and be captured into the `stdout` result attribute. `err_stream` and `stderr` will always be empty when `pty=True`.

- The official documentation is solid. Get familiar with it.

1.16 Logging Is a Programmer's Best Friend

Main point

You should log. No excuses.

- Logging makes the flow of the application obvious & visible. This helps every interested party to reason about what and when is happening.
- If done right, logging may literally save your day when real problems shred your beautiful code to pieces (especially in the production environment, where debugging turns into a literal nightmare).
- It can be tedious to configure the logging yourself. That is why in Big-Bang-py you can find pre-configured setup that is ready-to-go.

  - In proposed setup logs are streamed to stderr as well as saved in `$PROJECT_ROOT/logs`.
  - Logs saved on the drive are processed by RotatingFileHandler. Therefore there is no risk that log files will grow indefinitely.
  - How to get started?

```
# Load logging config
import logging.config
from src.logging_config import DICT_CONFIG
logging.config.dictConfig(DICT_CONFIG)

# Get logger
logger = logging.getLogger('main')

# Logging time!
logger.info('* THE GREAT BIRD IS LISTENING *')
```

Power-Hint!

`Flake8 + Pipfile` configuration of Big-Bang-py repo uses flake8-print plugin, which will find and warn you about using print statements. If they are no longer necessary, remove them. But if they are... log there! There is great chance that sooner or later you will need this information again.

- You can learn more about logging basics on Real Python.
  - If you are an adventurous programmer with no faint-heart, there is the official Python logging documentation waiting out there. Bear in mind that because of this document, and its unnecessary complexity, a lot of people were scared off and have been using prints ever since.

1.17 README - Gateway to Your Code

Main point

Write a README, because no one can read your mind (at least not yet).

1.17.1 The Why

1. Documentation forces you to prove that you understand how the application works and that you can explain it to the World. Your docs become your ultimate rubber duck!
2. By bringing the essential knowledge about the project in a structured manner, README serves as a lantern for all of the programmers from the future - including the future YOU!
3. We should be honest with ourselves and accept that over time code becomes more and more alien. In fact, dreadful legacy code is simply anything we're not writing right now. And README helps with legacy code A LOT!
4. We have all been there. So out of sheer respect to our countless hours lost trying to figure things out from the source, we should do our best to save the pain for the programmers to come.
5. If you are not convinced yet, bear in mind that writing docs is a discovery journey into the very depths of your app. And this usually results in finding most surprising bugs in most surprising places! :D

1.17.2 Content

There is no obvious consensus regarding what README should contain. However, it seems that a good one should **at least** answer the questions below (provided they are applicable):

- Project:
  - What is it about?
  - Why is it needed?
  - How does it solve the problem?
- Application/Library:
  - How to build, install & test that it works?
  - How to use it?
  - How to deploy?
- How to contribute to the repo?

If you want to get inspired, Make a README and Awesome README are fantastic sites to dig.

1.17.3 Format

The recommended format of README is Markdown. This lightweight markup language is simple, powerful and popular, making it a perfect choice for most of the projects.

A viable alternative is reStructuredText. However, it is far more advanced than Markdown, which for README might be an overkill.

1.18 Continuous Integration - Kill Bugs Fast

Volumes have been written about Continuous Integration (along with Continuous Delivery and Deployment), yet...

Main point

> The absolute minimum every project absolutely positively must implement is to verify each check-in to a hosted/shared version control by an automated tool. All in order to detect problems early.

- Keeping it more serious, CI will take a lot of mental burden off your head. You cannot assume that every contributor has correctly configured Pre-commit Git Hook. You cannot assume everyone will remember to test and lint. You cannot even trust yourself, because sooner or later you will also make some stupid mistake. That is why you hire CI - to always have your project's back. (Sometimes even for free!)

- There are numerous CI tools, both closely related to your hosting services (like Bitbucket Pipelines or GitLab CI/CD) as well as platform-independent ones (like Jenkins or Circle CI). Choose whatever floats your boat.
- **As to what to check during CI, at least run the tests to double-check they pass.** A handful of other ideas:

– Iron the new code with your linters of choice.

– Test if the project still builds correctly.