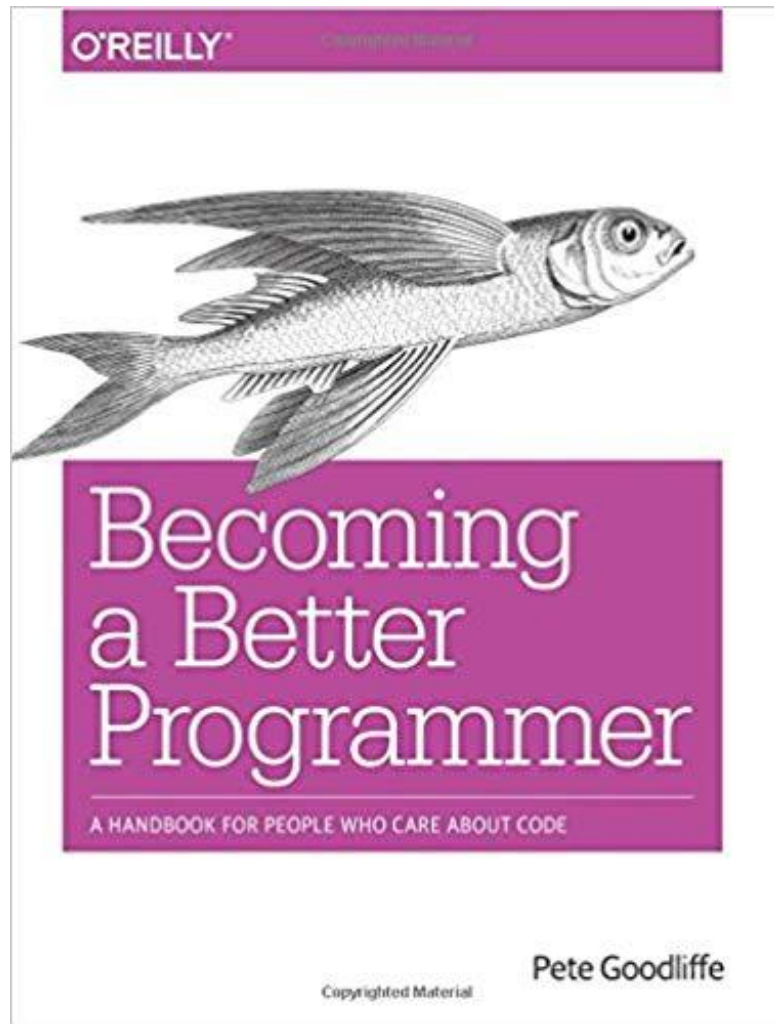# Software Best Practices
Effective Version Control

Alex Olivas
Madison Bootcamp 2020

The Zen of Python : When in doubt "import this"

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

The Zen of Python : When in doubt "import this"

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Readability counts.

Commit to a style guide
https://google.github.io/styleguide/pyguide.html

Use a tool to enforce a style, e.g. pylint.

Don't rely solely on documentation

General Rule: "Don't leave to documentation what the compiler or runtime system can enforce."

Example: GitHub Actions for Pylint

https://github.com/marketplace/actions/github-action-for-pylint

Effective Version Control

Central Collaboration Hub
Feeds CI, Release Engineering, Code Audit systems
Facilitates Software Archaeology
Provides Backups
Fosters Rhythm and Cadence (i.e. Workflow)
Enables Concurrent Development Streams

Getting Started w/ GitHub

Use it or Lose it!   https://github.com/

Effective Version Control

What to Store?

Two options

Store Everything

- Store **EVERY** file that's required to recreate your result.
- Source code, docs, build files, configuration files, assets, third party libs.
- Strive for 'turn-key' after checkout.
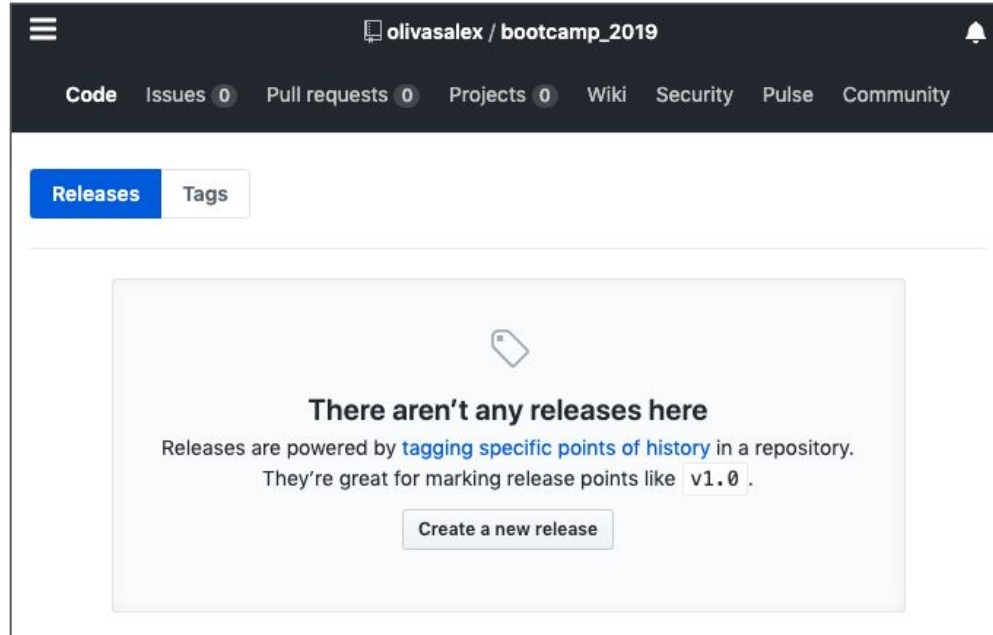
Store as Little as Possible

- Keep it slim and simple
- Exclude compiled code, cmake generated files, byte-compiled python, etc...
- No development tools or OS image

Do Both! They're not contradictory.

Effective Version Control

Making Releases - https://help.github.com/en/articles/creating-releases

Generating DOIs - https://guides.github.com/activities/citable-code/

Effective Version Control

Semantic Version Numbers - https://semver.org/

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Effective Version Control

Repository Layout

Invest time in a clear, easy to navigate layout.

```
README.rst
LICENSE
setup.py
requirements.txt
sample/__init__.py
sample/core.py
sample/helpers.py
docs/conf.py
docs/index.rst
tests/test_basic.py
tests/test_advanced.py
```

Prefer a standard layout to make it easy for others to navigate your project. Increases adoption.

For python projects prefer virtual environments
$ python3 -m venv my_virtual_env
$ source my_virtual_venv
$ pip install <some_lib>
$ pip freeze > requirements.txt
$ pip install -r requirements.txt

https://docs.python-guide.org/writing/structure/

Effective Version Control

Make Small, Atomic Commits
"Commit early, commit often."

- Make the commits small enough that they don't break the code. What constitutes "broken" code? - Doesn't compile. Tests don't pass.
- **DO NOT** commit something that covers more than one change: "git commit -m 'Refactor and critical bugfix' " BAD
- **DO NOT** wait until the end of the day or week to commit.
- **DO NOT** mix functional changes with whitespace cleanups.
- **DO** write good commit messages.
  - Good commit message: "Fixes issue #123: Use std::shared_ptr to avoid memory leaks. See C++ Coding Standards for more information."
  - Bad commit message: "blerg"

Effective Version Control
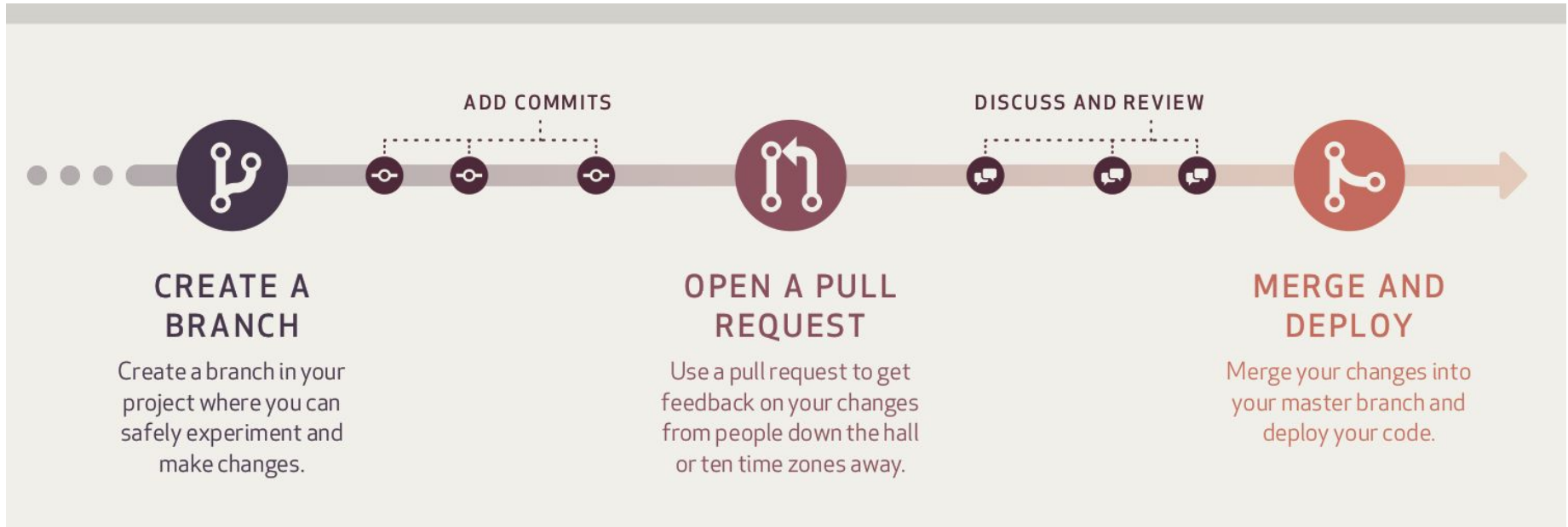
Make Quality Commits - **Don't break the build!!!**

1. Make change
2. Test that it builds against master/trunk
3. Ensure all the tests pass. **(Invest in a test suite!!! e.g. python unittest)**
4. Check it in w/ an **informative** commit message.
5. Check your continuous integration (CI) system.

Measure test coverage with pycoverge and display this in README.md

Effective Version Control

https://guides.github.com/introduction/flow/

ADD COMMITS

DISCUSS AND REVIEW

CREATE A BRANCH
Create a branch in your project where you can safely experiment and make changes.

OPEN A PULL REQUEST
Use a pull request to get feedback on your changes from people down the hall or ten time zones away.

MERGE AND DEPLOY
Merge your changes into your master branch and deploy your code.
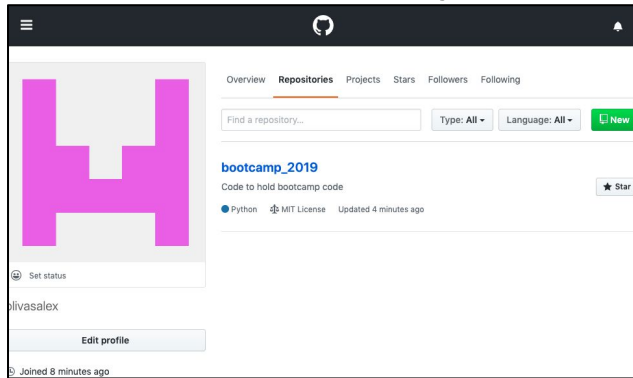
Exercise: Getting Started w/ GitHub

Use it or Lose it!   https://github.com/
Add example code from yesterday.

Click on **NEW** to create a new repository.

$ git clone https://github.com/<username>/<repo_name>
$ git add <filename>
$ git commit -m 'initial commit'
$ git push