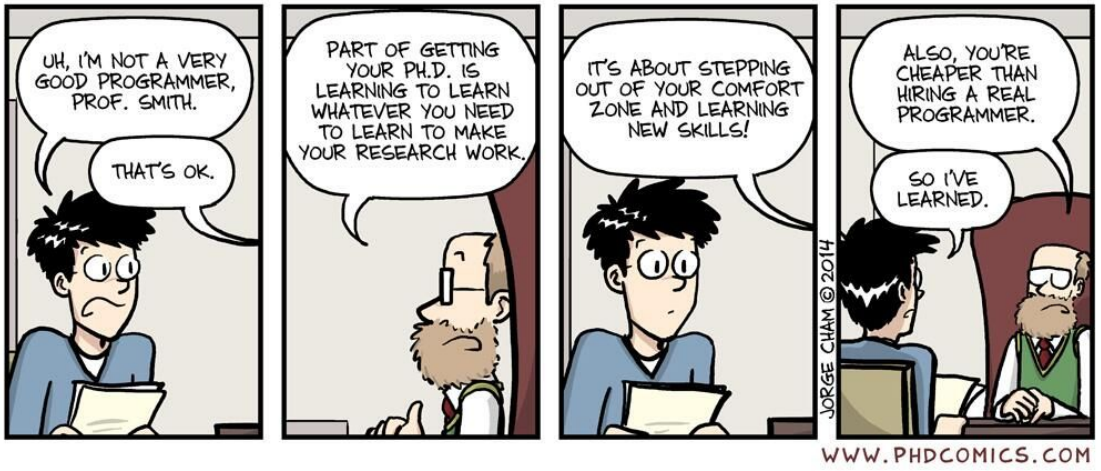
Practical Software Engineering

for Researchers (using Python)

Prerequisites

- Basic programming ability, understanding of procedural and object-oriented programming.
- Intermediate familiarity with Python.
- Having actually worked on a software project in one capacity or another would be helpful.



Software Crisis in Academia

Software Crisis in Academia

- There is a kind of software crisis in research in my opinion, similar to the software crisis in the early days of computers.

> The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.
>
> — Edsger Dijkstra, The Humble Programmer (EWD340), Communications of the ACM

- Similarly as more and more research relies on computers there comes a point where researchers education becomes insufficient to write software of acceptable quality.

How does it manifest itself?

- Lack of reproducibility: how many of you successfully ran or even thought about running the code associated with the papers you read? Assuming it is even available.
- Wasted productivity: it is not normal to spend 90% of your working time debugging the code a PhD student wrote 3 years ago. Or even your own code you wrote a week ago.
- Wasted time due to lack of automation of easily automatable tasks. Such as testing the code, publishing documentation or publishing of the software packages in an installable form.
- Embarrassment from publishing results that are due to problems and bugs in your code rather any real achievement. Similarly the unease in knowing someone might actually see your code some day.

A personal anecdote (I)

- In the early days of Covid I was working on a project called VECMA which had to do with uncertainty quantification. At one point a scandal broke out about a code that models spread of diseases that was developed at Imperial College London.
- The code seemed to be non-deterministic even though it shouldn't have been. So someone in the project suggested we quantify those uncertainties
- We tried to identify the source of them and it seems that this researchers code suffered from race conditions that meant that it produced different results every time.
- The code was several thousand lines of C++ all dumped into one large procedure with no structure.

A personal anecdote (II)

[Imperial College epidemiologist ****] ******** was behind the disputed research that sparked the mass culling of eleven million sheep and cattle during the 2001 outbreak of foot-and-mouth disease. He also predicted that up to 150,000 people could die. There were fewer than 200 deaths. . . .

In 2002, ******** predicted that up to 50,000 people would likely die from exposure to BSE (mad cow disease) in beef. In the U.K., there were only 177 deaths from BSE.

In 2005, ******** predicted that up to 150 million people could be killed from bird flu. In the end, only 282 people died worldwide from the disease between 2003 and 2009.

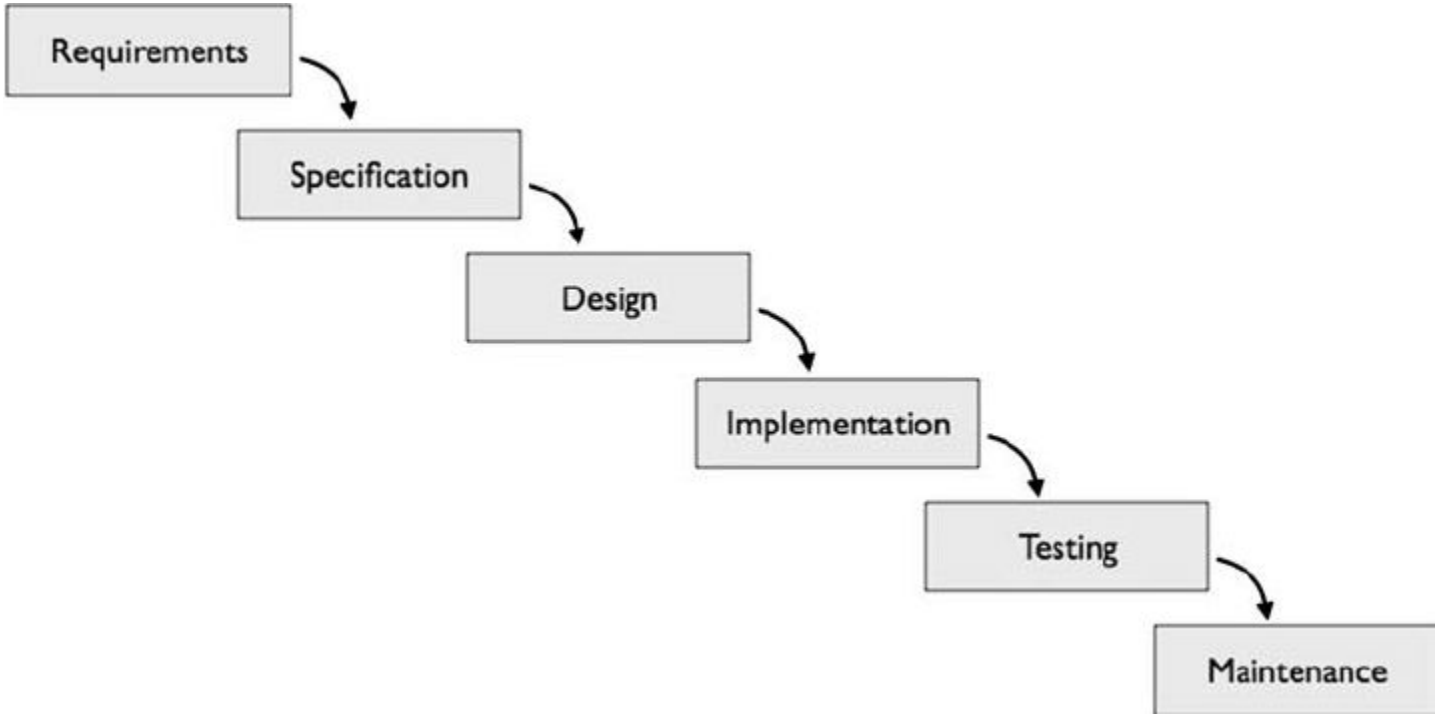
In 2009, a government estimate, based on ********'s advice, said a "reasonable worst-case scenario" was that the swine flu would lead to 65,000 British deaths. In the end, swine flu killed 457 people in the U.K.

Last March, ******** admitted that his Imperial College model of the COVID-19 disease was based on undocumented, 13-year-old computer code that was intended to be used for a feared influenza pandemic, rather than a coronavirus. ******** declined to release his original code so other scientists could check his results. He only released a heavily revised set of code last week, after a six-week delay.

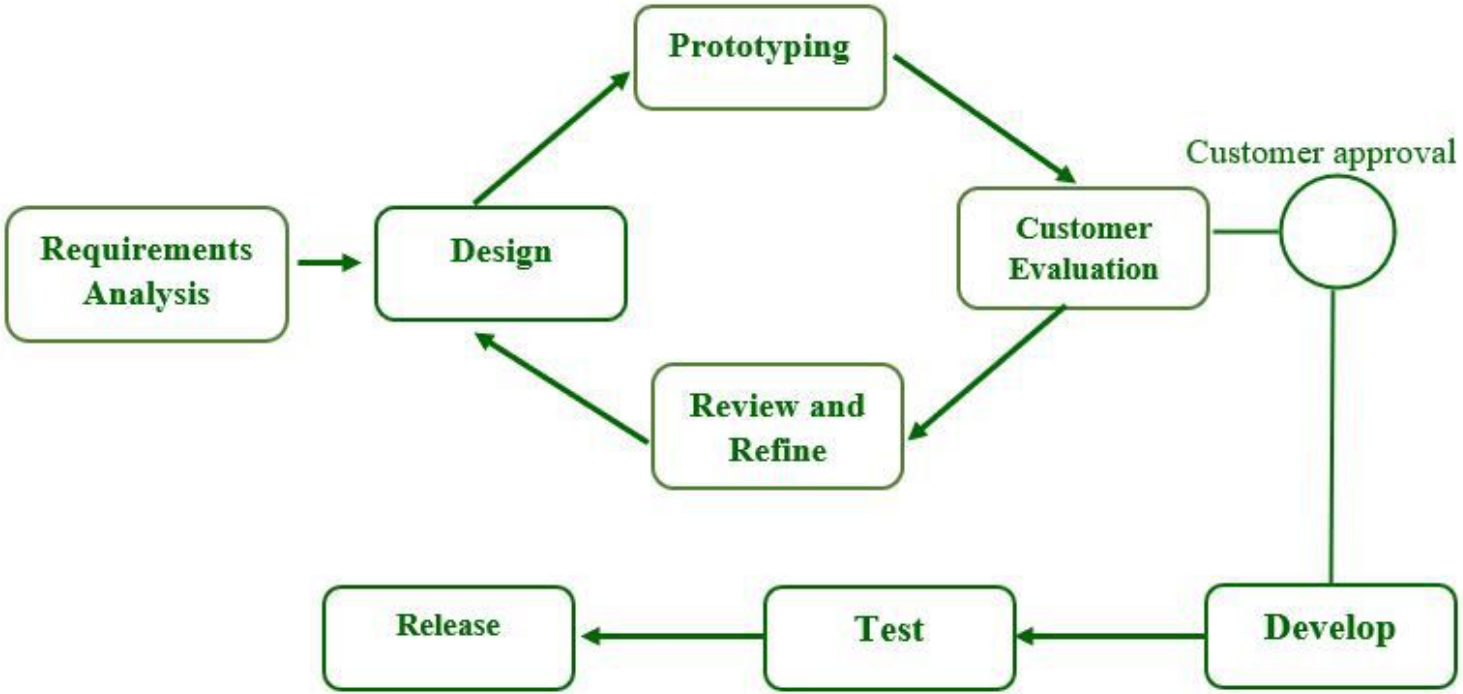
Software Engineering

- Software systems are the most complex things humanity has ever built.
- Failure of said systems costs time, money and sometimes lives.
- Software engineering aims to apply methods reminiscent of conventional engineering to software development.
- Software engineering informs the software development process.
- We will mostly be talking about the software development process. Or more specifically my own take on a software development process most suited for researchers whose primary focus is NOT software development.
- But first here are some common approaches to software development process...

The Waterfall Model



The Prototype Model



Formal Methods

- Attempts to prove correctness of program as if they were mathematical theorems.
- Usually programs are written with formal verification in mind in languages that support it.
- May be necessary when dealing with software for medical equipment and other cases where the cost of bugs is unacceptably high.
- Examples of technologies used for formal software verification:
  - https://coq.inria.fr/
  - https://wiki.portal.chalmers.se/agda/pmwiki.php

Agile Methods

- Usually based on a cyclical application of something similar to the waterfall model.
- Software is developed in short cycles at the end of each requirements are reviewed and software is changed accordingly.
- Common traits:
  - Strong reliance on automated testing (since the software is frequently changed)
  - Fairly flexible project management where tasks are created and updated during frequent meetings
  - Frequent review of requirements and gathering of feedback
  - Strange seeming practices like pair programming designed to address software engineering challenges in creative and ad hoc ways

What's Special About Academia and How Much of This Applies to Us?

- We don't have the profit motive so we can be a bit more careful and thoughtful about how we work (less time pressure and having to appease customer demands).
- The customer in this case is the scientific community and software should address the interests of said community. This applies even if you are not going to release your software.
- The emphasis when designing scientific software should be on:
  - Correctness, otherwise you risk embarrassment and possible career damage due to publishing of non-results.
  - Ease of maintenance, since scientific software needs to be maintained under very limited resources.
- Almost any methodology is better than none but I think some form of Agile probably offers least overhead and most payback in academic settings.

Lesson Plan

- Learn how to create, build and install a Python package.
- Learn about the GitHub Flow.
- Test Driven Development and testing with pytest.
- Documenting code.
- Publishing a Python package.
- Static code analysis tools.
- Design by Contract using Deal.

Every lesson also include elements of CI/CD since we are going to automate all the steps.

Baby's First Python Package

What I Think is the Current Way of Structuring a Python Project

Structure of a (Modern) Python Project

```
packaging_tutorial/
├── LICENSE (empty)
├── pyproject.toml
├── setup.cfg (empty)
├── README.md (empty)
├── src/
│   └── example_package/
│       ├── __init__.py (empty)
│       └── example.py
└── tests/
    └── test_main.py (empty)
```

Structure of pyproject.toml

```
[project]
name = "hello-world"
version = "1.0.0"
description = "My first Python package"
requires-python = ">=3.8"
authors = [
    {name = "John Doe", email = "john@example.com"},
]
dependencies = [
    "numpy",
    "pandas",
]
```

Building the Package

If your project is structured as above, then build it with:

```
$ python -m build
```

Will build the wheel which you can then install with pip install:

```
$ pip install dist/package_name.whl
```

Or you can do an editable install with

```
$ pip install -e .
```

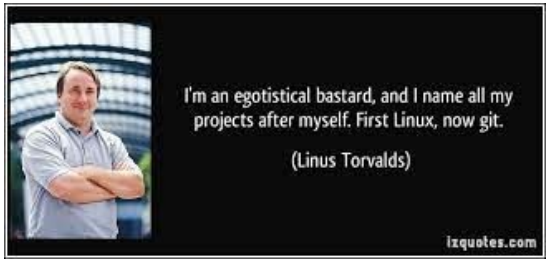
This requires the `setup.cfg` file to be present for some reason, but as far as I can tell it can be empty

Python Project Exercises

1. Create a skeletal Python project with a pyproject.toml file
2. Add a function to the package called hello_world that prints "Hello, World!"
3. Build the package
4. Install the package
5. Import the hello_world function from it and run it

The GitHub Flow

and Setting Up CI/CD



What is Version Control?

- Version control enables collaborative development because it:
  - Tracks who made changes in the code and what they are.
  - Allows you to revert back to a previous version of the code.
  - Allows parallel versions of the software to be developed at the same time and then merged to a single version later.
- Git (originally by Linus Torvalds) seems to have eliminated all other version control systems and is pretty much the only game in town.
- Git is decentralized because it doesn't rely on a central authority to determine the "correct" version of the software. Unlike older systems like SVN or CVS.
- People still tend to sometimes use git in a centralized way though. Especially in one or two person projects.

Version Control Vocabulary

`repository` – source code along with the full change history

`clone` - your own copy of the repository

`fork` - a copy of the repository that is being independently maintained

`branch` – a divergent change history that is maintained in parallel with other branches

`commit` - a smallest unit of change in the source code

`merge` - incorporate changes from one branch into another one

`conflict` - occurs when two branches have changes to the same file

`pull (merge) request` - a request to incorporate changes from a branch to the main branch of the project

`check out` - update your local copy of the repository

`rebase` - incorporate changes made in the main branch to your branch

Initialize Your GitHub Repository

```
git init -b main

git add .

git commit -m "first commit"

git remote add origin
git@github.com:yourusername/se-tutorial.git

git push -u origin main
```

Git Commands You Will Need the Most

To create a commit with your latest changes

```
git commit -am "what was changed and why"
```

To push your changes to the repository

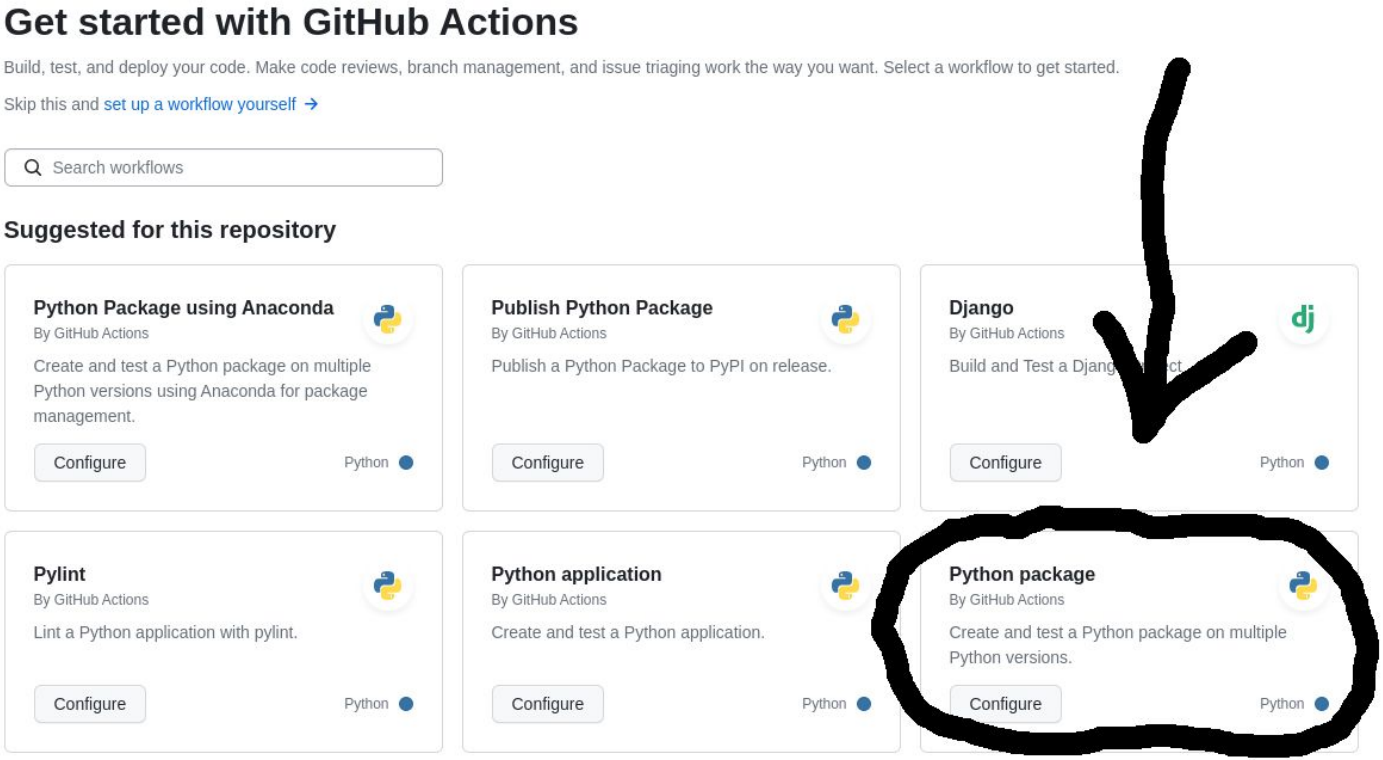
```
git push
```

Version Control Exercises

1. Create a GitHub project for this tutorial.
2. Initialize a git repository for the Python package we have developed in the previous exercise.
3. Push it to your GitHub.
4. Add a small change (such as editing the README.md file). Push the changes to the repository.

   NOTE: You will need to set up authentification on GitHub correctly. Right now easiest seems to be through the use of an SSH key.

Setting Up CI/CD for Your GitHub Repository (I)



Setting Up CI/CD for Your GitHub Repository (II)

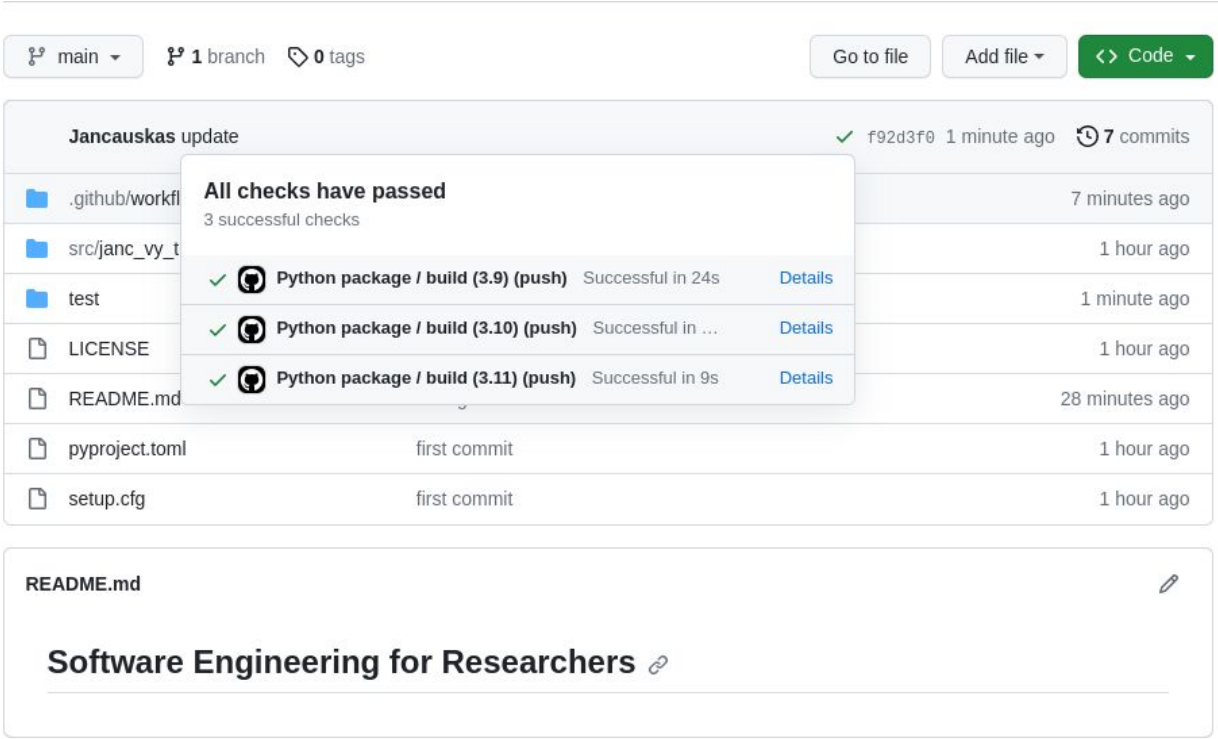
- This will create a new directory called `.github/workflows` in which you will put your CI/CD scripts in YAML format
- Change the workflow file so that the `pytest` command instead says `pytest test/`
- Under test/test_main.py add a simple test that always succeeds otherwise the pipeline will fail if it finds no tests:

```
def test_main():
    assert(True)
```

CI/CD Set-Up Exercises

- Enable CI/CD for your project (via Actions menu in GitHub)
- Run pytest with CI/CD after each push on the `test` directory
- Add a test to `test_main.py` that always succeeds
- Afterwards test that the test executed successfully for all the Python versions
- Investigate the execution log

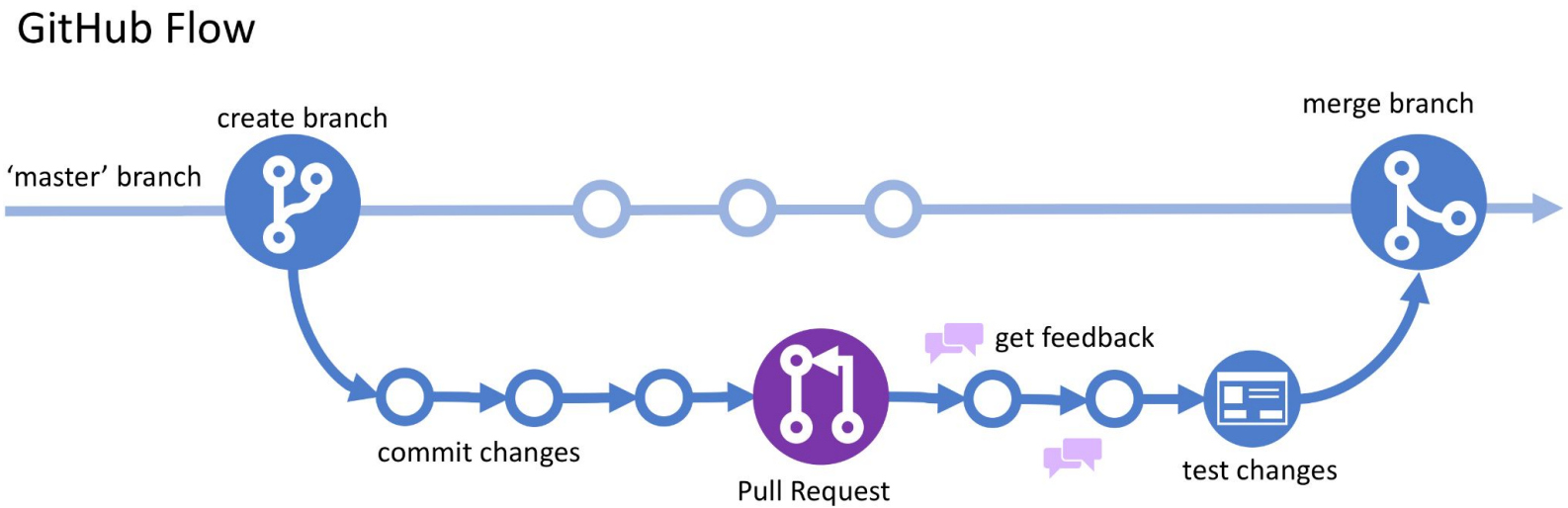
What Success Looks Like



More On That Workflow File

```yaml
name: Python package

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        python-version: ["3.9", "3.10", "3.11"]

    steps:
    - uses: actions/checkout@v3
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
      with:
        python-version: ${{ matrix.python-version }}
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        python -m pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
    - name: Test with pytest
      run: |
        pytest test/
```

What is the GitHub Flow?



GitHub Flow Step 1: Create a Branch

Both GitLab and GitHub let you do this from the web interface but we will use command line since that is universal:

```
$ git checkout -b branch-name
```

The `-b` option creates a new branch and switches to it. To switch to an existing branch:

```
$ git checkout branch-name
```

GitHub Flow Step 2: Make Commits to Your Branch

Commits should be relatively small. After each commit your software should still work (that is you should not make commits that break stuff, such as changing a method name and not updating that name where the method is called). Multiple file commits are generally discouraged and a single commit should be able to fit in your own memory. Commit messages should describe what was changed, where and why.

```
$ git commit -am 'message'
```

Good Commit Message Examples (from NumPy)

```
* BUG: fix for modifying the index arg in ufunc_at
* TST: skip FP exceptions test on 32-bit Windows
* DOC: Fix some incorrectly formatted documents
* DEV: Use exec_lines and not profile dir for spin ipython

`spin ipython` is annoying me by using a temporary profile dir which
means that every invocation has its own history file.

Trying things out, it seems like `--TerminalIPythonApp.exec_lines=`
is a safe way to achieve the same thing (import numpy before anything
else) while not interfering with `-c "code"` (or even more exec lines
statements).
```

Long Commit Message

The last example was of a long commit message. These cannot be added with the `-m` option and need to be edited with a text editor. To this end simply omit the message option and you will be taken to the text editor.

```
$ git commit -a
```

GitHub Flow Step 3: Push Your Branch

When pushing your branch for the first time:

```
git push --set-upstream origin small-change
```

After that you can check which branch you are in with:

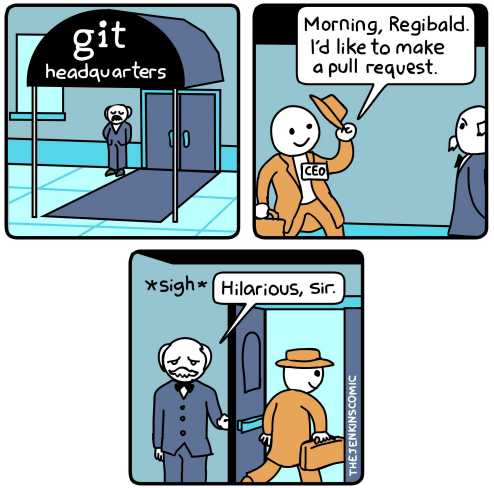
```
git branch
```

And push changes with:

```
git push
```

GitHub Flow Step 4: Make Sure Tests Pass

- Add tests for the new functionality.
- Make sure existing tests pass.
- Document your changes. Make sure documentation builds.



GitHub Flow Step 5: Make a Pull Request

A pull (GitHub) or merge (GitLab) request is a request to merge your branch into the main project branch. While there seems to be a way to do it from the command line most people will probably use a corresponding function in either GitHub or GitLab.

- Pull requests can be made very early. This allows other project members to track progress. In fact this is usually a good idea. Early pull requests are often marked as "Draft".
- Pull requests should be reviewed by other project members who then can leave comments, often on a line of code basis.
- Pull requests should only be merged if all tests succeed.

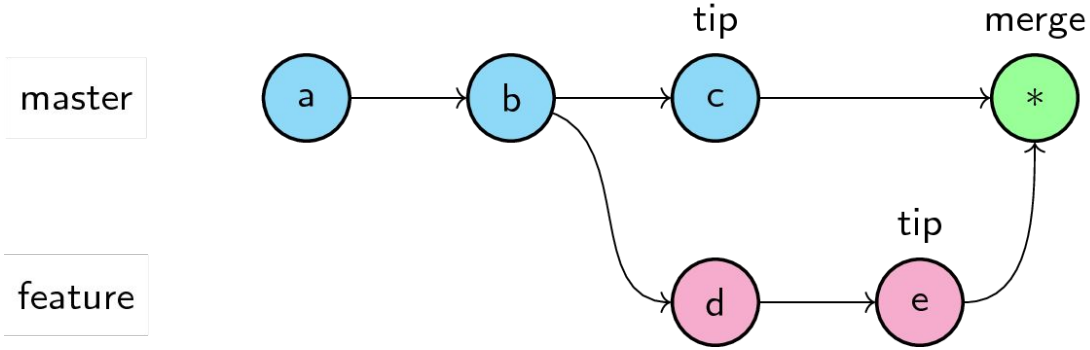
GitHub Flow Step 6: Get Feedback on Pull Request

DLR

- Other project members can review and comment on your pull requests on both GitHub and GitLab based platforms.
- Reviewers can be assigned whose clearance must be obtained before the pull request can be merged.
- If changes are requested then those requests can be addressed by simply making changes to the branch in question and pushing them to the remote repository from which the pull request was made.

GitHub Flow Step 7: Merge Changes in to the Main Branch

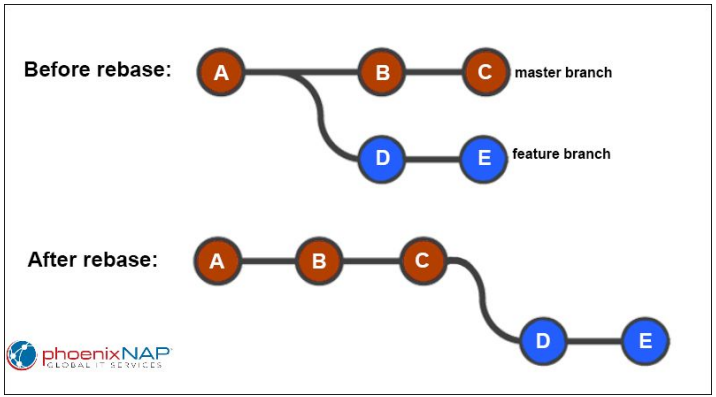
- Once everyone is happy changes can be merged
- Any resulting conflicts will need to be resolved by hand
- These arise when the same files have been changed in the main branch and in the branch of the pull request



Rebasing

- It will often occur that you are working on a large Pull Request over a longer period of time.
- During that time changes might be made to the main branch that are incompatible with changes you've made.
- You might consider rebasing in those cases.
- Think of rebasing as moving the moment you created the branch into the future.

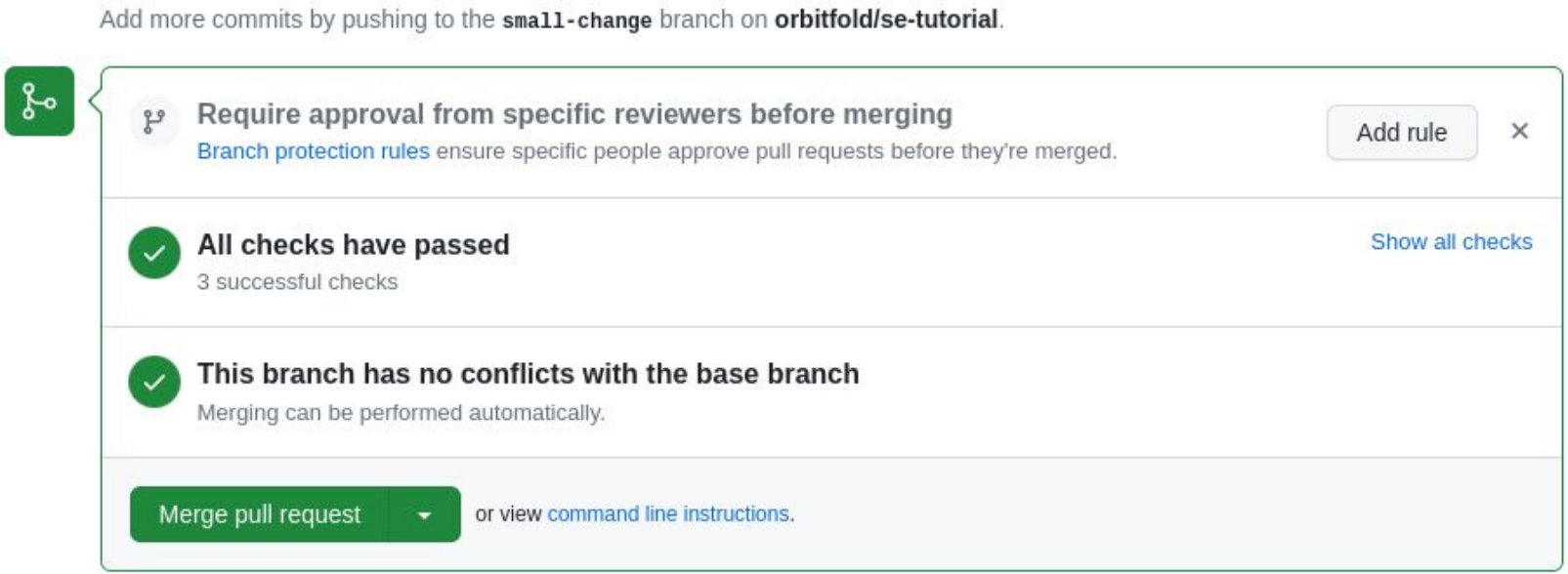
```
git rebase main
```



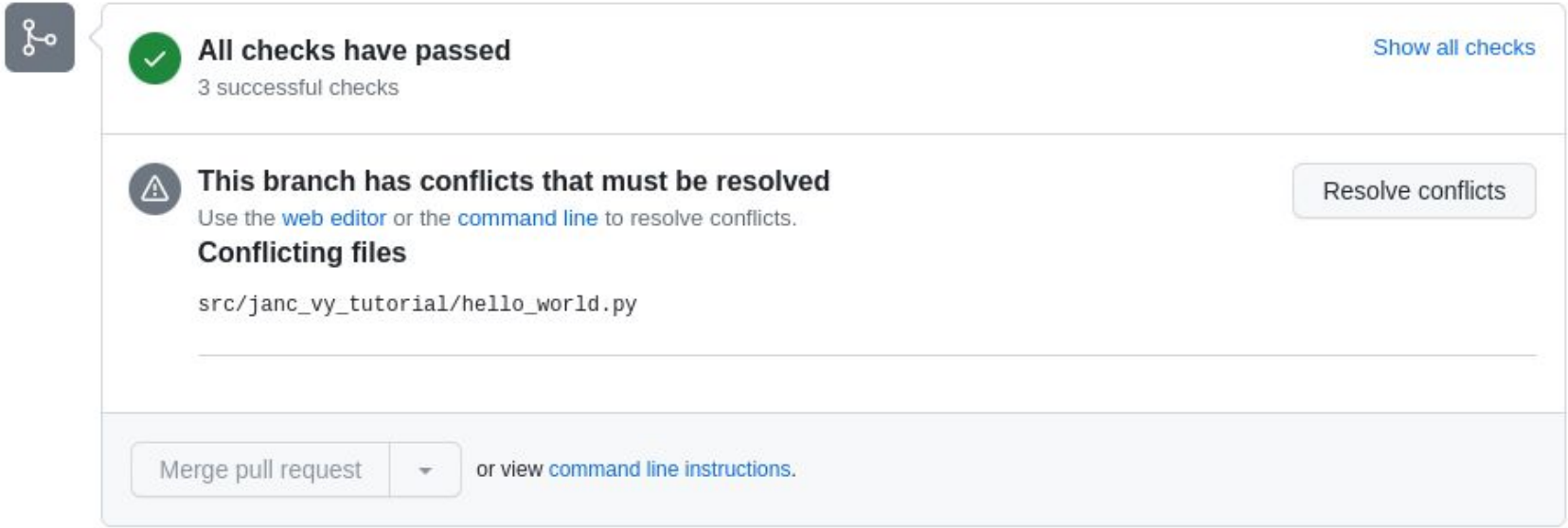
GitHub Flow Exercises

- Create a branch introduces changes to the branch
- Create a pull request
- Leave a comment on the pull request code
- Introduce conflicting changes in the main branch
- Merge pull request solving the conflict

Pull Request with Tests Passing and No Conflicts



Pull Request with Conflicts



Conflict Resolution

src/janc_vy_tutorial/hello_world.py

```
1  def hello_world():
2  <<<<<<< small-change
3      print("Hello, World!!!")
4  =======
5      print("!!!")
6      print("Hello, World!")
7  >>>>>>> main
8
```

Test Driven Development

Three Rules of Test Driven Development (TDD)

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Test Driven Development

Ideas Behind TDD

- Utilize tests to:
  - Aid in the design stage: because you are writing tests in advance you need to think through the requirements for your software, interfaces, etc.
  - Aid in documentation: tests serve as a form of documentation for what the software is supposed to do.
  - Aid in refactoring: tests make up a safety net which allows you to change software with greater confidence.
  - Aid in understanding: failing tests allow you to understand what effect your changes have on the software.
  - Aid in testing since developers double as testers.
  - Aid in making sure tests are actually written (if tests are not mandatory no one writes them)

Ideas Behind Test Driven Development

- The main idea behind Test Driven Development (TDD) is to write tests before implementation.
- The implementation is then checked against these tests. The tests can be further refined during the implementation stage.
- This may sound backwards, but:
  - It allows you to refine your understanding of the problem.
  - It allows you to think through the interface and structure of your code.
  - Unless you write tests early there is a good chance they will never be written.
- If you are to ignore the rest of software engineering practices I personally believe that TDD would help you write your research code in the shortest amount of time possible if your software is relatively small in size.

Unit vs. Integration Testing

- Unit tests are meant to test that individual units of software (methods, classes, etc.) work as intended.
- Integration testing tests the whole software or it's modules as a whole.
- Integration tests test interface between larger modules, treating the module as a black box.
- As an example, if you have a command line program, integration testing will run the full program with various arguments and test whether it produces the results one would expect of it.

Testing in Python

Currently the most commonly used testing framework in Python seems to be pytest and for very good reasons:

- Minimum amount of boiler-plate code - in fact none if your tests are relatively basic.
- Well integrated into the Python ecosystem - in fact a lot of tools expect that you will be using pytest.
- Incredibly powerful if you delve deeper into it. Supports various helper tools for test automation.

Main pytest concepts

- Put tests under the `test` subdirectory in your source tree.
- Tests are simply Python files that start with `test_*`.
- Test cases are functions inside those files whose names begin with `test_*`.
- Running the tests is as simple as typing `pytest` in the root folder of your source directory.
- Testing is done with the `assert` statement. Pytest is clever enough to provide you useful information when `assert` fails.
- Please note that `assert` in the context of Pytest does a lot more heavy lifting than normal `assert` statements which merely raise an exception when the argument is `False`.

Basic Pytest Test Structure

```
def test_method1():
    ...
    assert(x == y)
    ...

def test_method2():
    ...
    assert(z in lst)
    ...
```

Using Fixtures

- Fixtures are resources shared between tests
- If you have a large data set or image, for example, you want to load it once and not in every test that uses it.

```
@pytest.fixture
def big_image():
    image = read_large_image()
    return image


def test_something(big_image):
    do_something(big_image)
    assert(something_succeeded)
```

Teardown Using Fixtures

- If the resource associated with the fixture needs to be cleaned up (e.g. some files deleted) you should use `yield` instead of `return`

```
@pytest.fixture
def big_image():
    image = read_large_image()
    yield image
    cleanup(image)

def test_something(big_image):
    do_something(big_image)
    assert(something_succeeded)
```

The tmp_path fixture

- This is a built-in pytest fixture that in my experience gets used all the time.
- Use it to store temporary files required for your test.
- Necessary if you code creates files. Store them under tmp_path and then test that they are created correctly.

```
def test_something(tmp_path):
    write_file(os.path.join(tmp_path, 'myfile.tiff'))
```

Pytest Parametrize

Parametrize lets you use a single test as if it were multiple tests by iterating over a list of arguments.

```
import pytest


@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Dealing With Randomness

- Testing functions that rely on randomness requires care since non-deterministically failing tests pose a big problem.
- There are essentially 2 possibilities:
  - Set the random number generator seed to a known value before testing the output.
  - Rely on the properties of the distribution of the output if it is well understood.

```
def test_random():
    np.random.seed(1234)
    x = np.random.random()
    assert(x == 3)
```

Mocking

- Mocking is creating place-holder objects that act as real objects.
- To understand why this is necessary consider a case where your software needs to connect to a service (e.g. Google Earth Engine). Inside the test environments the necessary credentials will not be available or at least very hard to set-up.
- You can use a mock object that acts like the Google Earth Engine.
- Similarly for database servers, Kubernetes clusters and many other similar things.

MagicMock

- Mock is a package distributed with Python. It allows for very easy mocking of methods.

```
from unittest.mock import MagicMock


thing = ProductionClass()
thing.method = MagicMock(return_value=3)
thing.method(3, 4, 5, key='value')
```

What Makes a Good Unit Test

- You should test corner cases:
  - If your method takes a list as an argument you should try always test how it behaves with an empty list.
  - In general all the lowest, highest and other "degenerate" cases should be tested.
  - Empty lists, dictionaries, zeros, etc.
- You should not access external resources. Use mocking instead.
- Unit tests must be deterministic. Even small probability influences should not (ideally) be allowed).
- You should not duplicate implementation. That is your test should not test against a different implementation of the thing you are testing.

Unit Testing Quiz

- Suppose we are writing a test to make sure a method that sorts a list in ascending order works correctly.
- What questions should our test ask?
- What are lists should we test it with?

```
def my_sort(lst):
    ...
    return sorted_lst
```

Basic Test

```
import pytest
from mypackage.sort import my_sort


@pytest.mark.parametrize("lst", [[], [1], [4, 5, 6], [6, 5,
4], [1, 2, 1]])
def test_my_sort(lst):
    sorted_lst = my_sort(lst)
    for x1, x2 in zip(sorted_lst[:-1], sorted_lst[1:]):
        assert(x1 <= x2)
    for x in sorted_lst:
        assert(x in lst)
    assert(len(lst) == len(sorted_lst))
```

Test-Driven Development Exercises

1. Write tests for a method(s) that (for example):
   a. Compute the integral of a function numerically (e.g. trapezoidal rule)
   b. Compute the integral of a function using some Monte Carlo method
   c. Find the largest element of a list using binary search
   d. Given two strings find the longest common substring.
   e. Given two strings, write a program that outputs the shortest sequence of character insertions and deletions that turn one string into the other.
   f. Multiply two matrices.
2. Use parametrization to test multiple examples of input parameters and output values.
3. Write the methods themselves, periodically running tests until the tests pass. Before running tests do an editable install with

```
pip install -e .
```

Documenting Your Code

with docstrings and pandoc

Docstrings

- Docstrings are the standard way to document your code in Python.
- Docstrings are multi-line strings that follow the method or class header. They explain the method, it's arguments and help users of your package understand what it does and how to call it.
- There are standards for how to write them and I usually use the NumPy one.

NumPy Docstring Style

```
def add(a, b):
    """The sum of two numbers.
    Parameters
    ----------
    a : int
        First argument to be added
    b : int
        Second argument to be added

    Returns
    -------
    int
        The sum of `x` and `y`
    """
```

Building Your Documentation

- Most common tool for building documentation is Sphinx but I wouldn't recommend using it to my worst enemy.
- A light-weight option I quite like is pdoc, although for more complex projects Sphinx may be the only option.
- To build documentation using pdoc:
  - `mkdir doc`
  - `pdoc src/yourmodule -o doc`

Publishing Documentation with GitHub Actions

- With pdoc it is very easy.
- Simply copy over https://github.com/mitmproxy/pdoc/blob/main/.github/workflows/docs.yml to your workflows folder.
- Edit it based on the comments in that file.
- Your documentation will now be at: https://username.github.io/reponame/

Documentation Building Exercises

- Add docstrings to your code from previous exercises.
- Build documentation using pdoc and open it in your browser.
- Add a GitHub action to automatically update and host documentation when a push is done to the repository or a pull request is merged.

Publishing Your Package

on PyPi

Publishing Your Package

- This part will serve two purposes:
  - Learning about the CD (Continuous Delivery) in CI/CD.
  - Make your software publicly accessible.
- Packaging software is usually time consuming and error prone if done by hand.
- We will learn how to do it automatically.
- The goal is to have a package be delivered automatically upon each release of your software.

Manually Publishing Your Package

You need a PyPI account for this. PyPI stands for Python Package Index and is the repository used by the pip package manager.

Make sure build and twine are installed:

```
$ pip install twine build
```

To test:

```
$ python -m build
$ python -m twine upload --repository testpypi dist/*
```

And then for real:

```
$ python -m twin upload dist/*
```

Publishing Your Package with GitHub Actions

```yaml
# This workflows will upload a Python Package using Twine when a release is created
# For more information see: https://help.github.com/en/actions/language-and-framework-gui

name: Upload Python Package

on:
  release:
    types: [created]

jobs:
  deploy:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.x'
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install setuptools wheel twine
    - name: Build and publish
      env:
        TWINE_USERNAME: ${{ secrets.PYPI_USERNAME }}
        TWINE_PASSWORD: ${{ secrets.PYPI_PASSWORD }}
      run: |
        python setup.py sdist bdist_wheel
        twine upload dist/*
```

CD Exercises

- Publish the package manually using twine.
- Set-up a CD pipeline on your GitHub repository that will automatically publish the package to PyPi when a release is created.
- Make a release, test the published package by installing it with `pip install`.

Static Code Analysis

Static Code Analysis Tools

- Static code analysis tools are used to enforce certain simple quality standards on your code.
- For example they can check that the code is formatted according to a chosen coding style.
- Some can also check for common simple bugs in your code.
- Other common usage is to calculate the test coverage of your code (which lines are covered by tests and which are not).

Linters

- Linters do some basic static checks of your code and are able to catch simple bugs and give warning (for example if a function is imported but not used).
- Pylama is a project that encompasses many such tools: `https://github.com/klen/pylama`
- Install it with `pip install pylama`
- Rut it with `pylama/src_dir`

Coveralls

- Coveralls is a service integrated with GitHub that gives you test coverage feedback.
- `https://coveralls.io/`
- It also lets you display a badge on your GitHub repository showing your test coverage percentage (as a way to show people that you've taken this tutorial and know the value of testing)

Static Code Analysis Exercises

- Add a pipeline to your repository that runs `pylama`
- Add `coveralls` to your repository

Design by Contract

A slightly stricter approach to Python programming

The Good Old Assert

The use of assert to check method arguments is a rudimentary form of design by contract. And if used extensively it can greatly help you debug your code.

```
def int_divide(a, b):
    assert(isinstance(a, int))
    assert(isinstance(b, int))
    assert(b != 0)
    return a / b
```

Classic Design by Contract

Design by Contract (DbC) is an extension of the idea of using `assert` to check correctness of programs. Programs designed using DbC use the following three checks:

1. Precondition - needs to be satisfied before the method is executed
2. Postcondition - needs to be satisfied after the method is executed
3. Invariant - needs to be satisfied at all times during the programs execution

Design by contract helps bring some of the advantages of static languages like C++/Haskell etc. to a dynamic language like Python.

Deal

- We will be using `deal` https://deal.readthedocs.io/ which adds DbC functionality to Python.
- There was a PEP that proposed to add DbC to Python proper. But at the time of this tutorial it hasn't been implemented.
- Install `deal` with `pip install deal`.

deal.pre (Precondition)

```
@deal.pre(lambda *args: all(arg > 0 for arg in args))
def sum_positive(*args):
    return sum(args)


sum_positive(1, 2, 3, 4)
# 10


sum_positive(1, 2, -3, 4)
 # PreContractError: expected all(arg > 0 for arg in args) (where args=(1, 2, -3, 4))
```

deal.post (Postcondition)

```
@deal.post(lambda x: x > 0)
def always_positive_sum(*args):
    return sum(args)

always_positive_sum(2, -3, 4)
# 3

always_positive_sum(2, -3, -4)
 # PostContractError:
```

deal.ensure (Postcondition)

```
@deal.ensure(lambda x, result: x != result)
def double(x):
    return x * 2


double(2)
# 4


double(0)
 # PostContractError: expected x != result (where result=0, x=0)
```

deal.inv


```
@deal.inv(lambda post: post.likes >= 0)
class Post:
    likes = 0

post = Post()

post.likes = 10

post.likes = -10
# InvContractError: expected post.likes >= 0

type(post)
 # deal.core.PostInvarianted
```

Deal Example

```
import math
import deal

@deal.post(lambda result: all([a <= b for a, b in zip(result[:-1], result[1:])]))
@deal.ensure(lambda lst, result: len(lst) == len(result))
def my_sort(lst: list[int]) -> list[int]:
    if len(lst) < 2:
        return lst
    mid = int(math.floor(len(lst) / 2))
    c = lst[mid]
    lst = lst[:mid - 1] + lst[mid:]
    a = [x for x in lst if x <= c]
    b = [x for x in lst if x > c]
    return my_sort(a) + [c] + my_sort(b)
```

Contracts in Production Code

- Contracts can be disabled in production code with `deal.disable()`
- This is similar to how `assert` is usually disabled in production code
- This is done because contract checking would otherwise be really resource intensive.

CrossHair

- CrossHair (https://crosshair.readthedocs.io/) is a kind of middle ground between testing and formal verification.
- To put it very simply it will take a program that has contracts written in some form (for example with deal) and will try to come up with counter-examples (inputs such that those contracts are violated).
- The functions also need to include Python type annotations: https://docs.python.org/3/library/typing.html
- Install CrossHair with `pip install crosshair-tool`
- Run it with `crosshair watch src_directory`
- You can read about the ideas behind it here: https://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf

DbC Exercises

- Add contracts to your code.
- Run CrossHair on it.

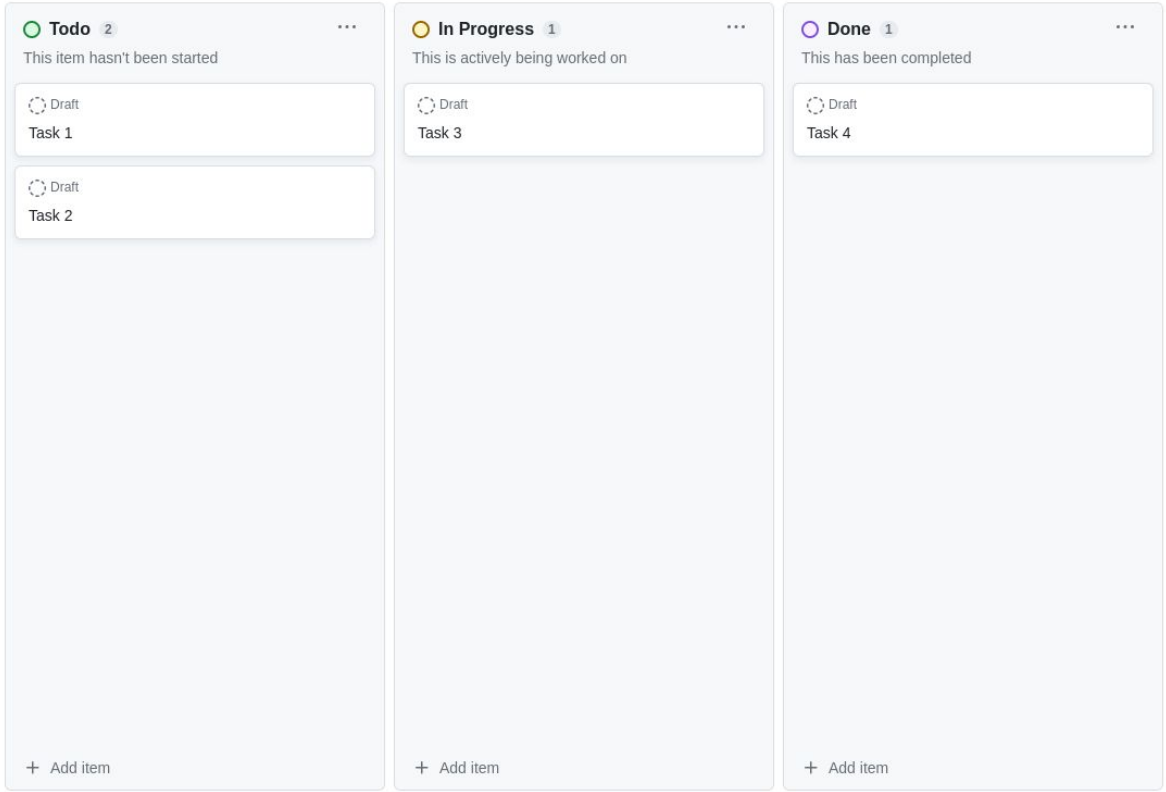
Basic Project Management

with Kanban

Kanban

- Kanban is a method to visualize the state of the project popularized by Taiichi Ohno from Toyota.
- Tasks are symbolized by cards placed on a board. Cards describe the task, who it is assigned to, it's priority and deadline.
- Tasks are placed into columns on the board which can be customized. In software projects the most common columns are "To Do", "In Progress" and "Done".
- The cards are moved from left to right depending on their status.
- This allows for a quick overview of the state of the project.
- A board is usually created for a relatively self contained project. For example a new feature.
- Kanban is a staple of many Agile methodologies.

GitHub Projects (I)



GitHub Projects (II)

- The Projects functionality in GitHub is closely integrated with the rest of GitHub.
- You can reference Pull Requests and Issues from the Issue tracker by number and links to them will be added automatically.

Kanban Exercises

- Create a project in GitHub.
- Create some items.
- Add links to Pull Requests or Issues inside the items.

"Since it's Christmas, I turned our Kanban Board into a Christmas Tree. Let's see if the Agile teams can figure out which items to work on next."

Some Random Tips

How to Debug From Command Line

Instead of debugging with `print` statements please consider using the built-in Python debugger to inspect the state of your program at any point.

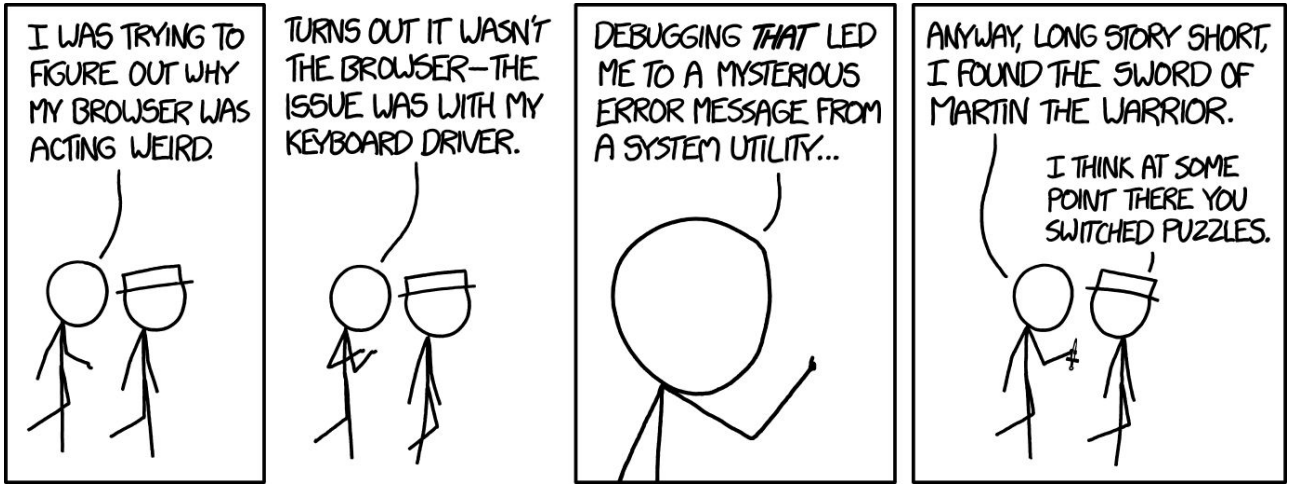
- Insert `import pdb; pdb.set_trace()` at any point in your code.
- When the software is run from command line you will be put in the debugger.

Useful Things to Do In the Debugger

- You can print variable values with `p variable_name`
- You can see the surrounding code with `list .`
- You can advance to the next line with `n`
- You can advance to the next trace point with `c`
- You can see other options with `?`

Debugger Exercises

- Insert a trace point at some point in your code.
- Run your program until you are put in the debugger.
- Try out the commands in the previous slide.



The End

Feedback Please

Feedback

- What did you find particularly useful?
- What did you already know?
- What was missing and you'd like included?