Continuous Data Science

Environments, Testing, and CI/CD

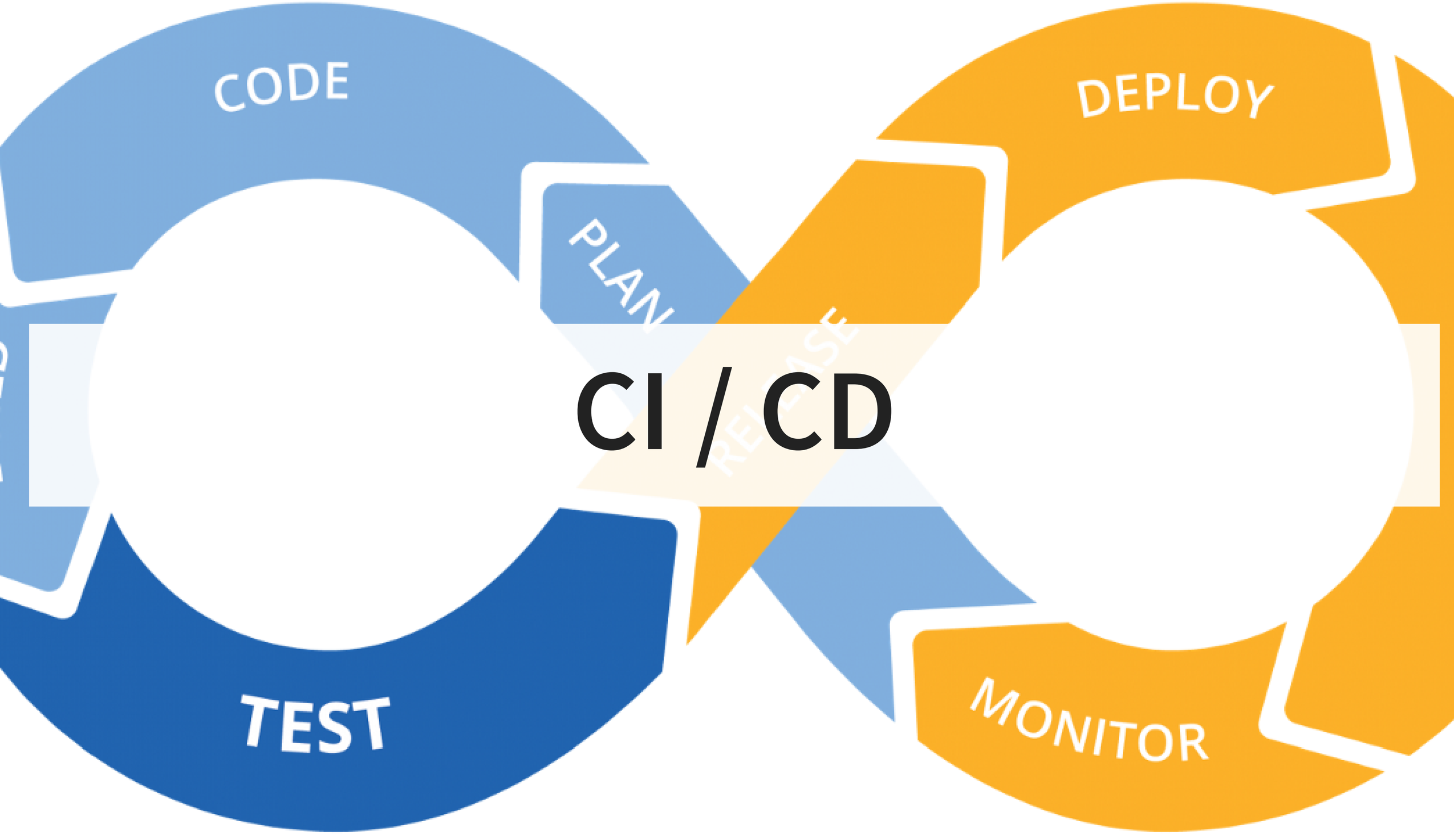
Dr. Scott Gorlin

Harvard University

Fall 2018

Agenda

- CI / CD
- What is *Python*?
- Testing
- Test Driven Development
- Testing tools
- Workflows
- Bootstraping
- Declarative State
- Pset 1
- Readings


CODE
PLAN
DEPLOY
TEST
MONITOR
CI / CD

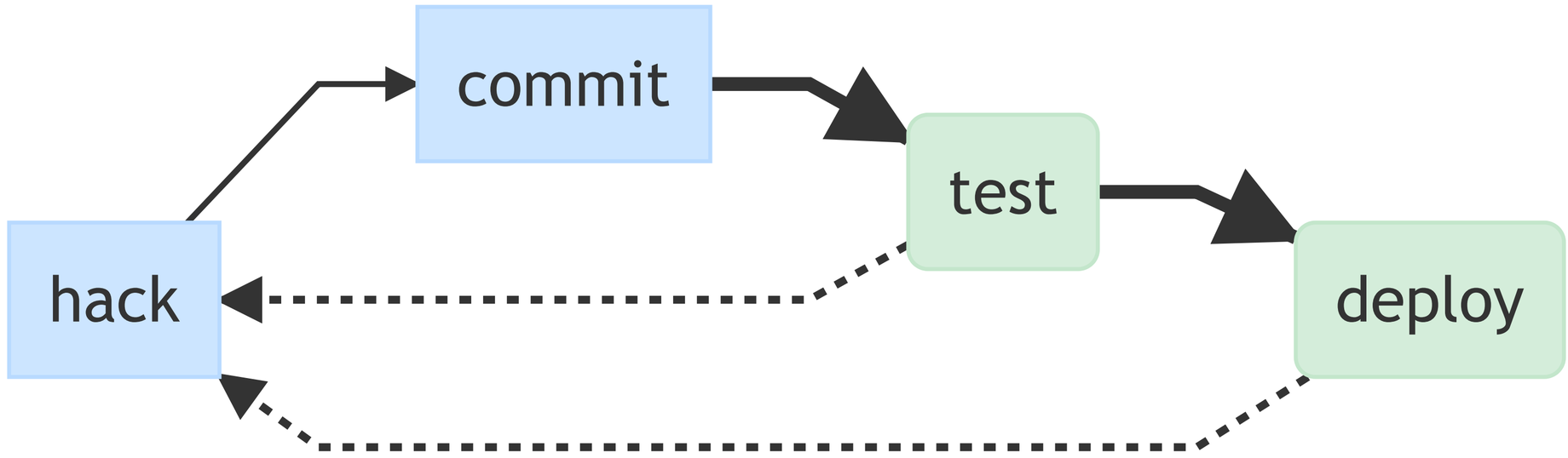
Continuous Science

CI/CD

- Rapidly merge code
- If tests pass, deploy
- Repeat

Science!

- Repeatable envs
- Repeatable data
- Tested algorithms



Software testing

Unittests

- Fast
- Extensive
- Safe and Stateless (don’t change your data)
- Reliable

Integration Tests

- Test other systems (eg, reading your DB)
- May be end-to-end
- Run on real input?
- Shadow deployments

. . . BUT SCIENCE IS SPECIAL

- More complicated environments
- Stateful, historical data
- Persisted results
- Probabilistic algorithms (random failures)
- ‘Qualitative’ results (eg, is model accuracy ***good enough***?)
- Evolving input data schemas
- Hard to get representative data
- Integrated, not isolated - relies on many services

Strategy

Repeatability

Environment and data are specified

Atomicity

We succeed or fail cleanly

Idempotent

Conform env/data to an expected state

Evolvability

Automatically upgrade data

Integration

Deployable in staging, prod, …

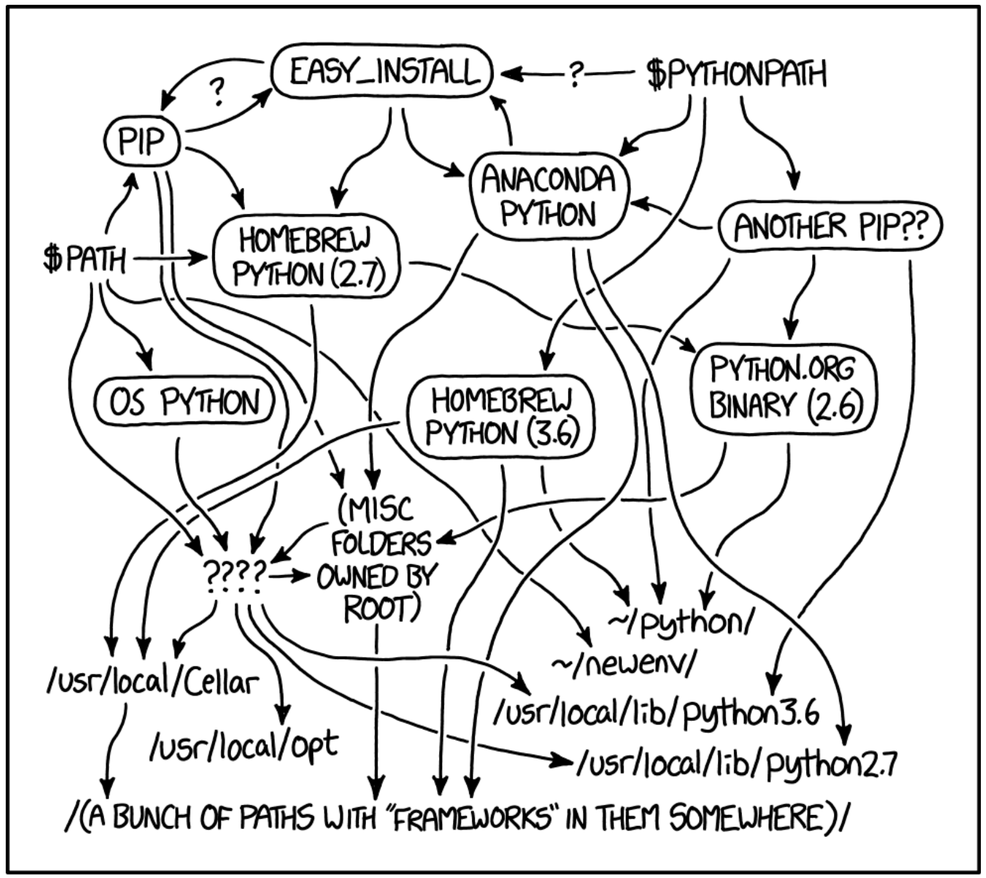
Auditable

Results tie back cleanly to their code

```
def primes(int nb_primes):
    cdef int n, i, len_p
    cdef int p[1000]
    if nb_primes > 1000:
        nb_primes = 1000

    len_p = 0  # The current number of elements in p.
    n = 2
    while len_p < nb_primes:
        # Is n prime?
        for i in p[:len_p]:
            if n % i == 0:
                break
        # If no break occurred in the loop, we have a prime.
        else:
            p[len_p] = n
            len_p += 1
        n += 1

    # Let's return the result in a python list:
    result_as_list  = [prime for prime in p[:len_p]]
    return result_as_list
```

What is *Python*?

You likely have 3



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

XKCD 1987

... EVEN IGNORING 2.X VS 3.X

System python	**Never, ever, touch**
Package managers/homebrew/pyenv	**Avoid manual pip**
└ **Virtual envs**	App-specific python-only envs
(Ana)conda	Better scientific stack
├ **Virtual envs**	Historically have had issues
└ **conda envs**	App-specific envs
Docker	Complete isolation

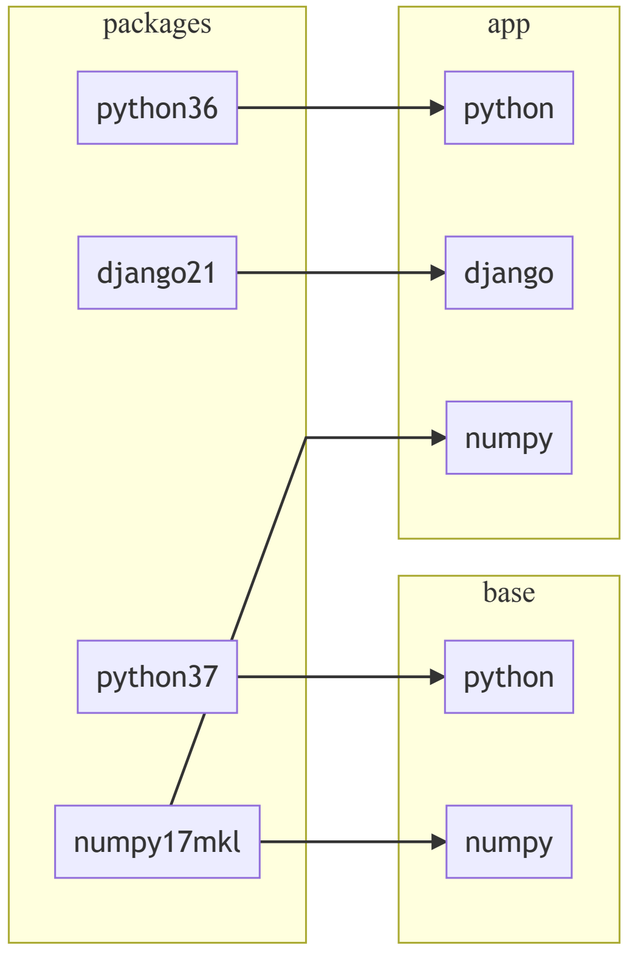
What's wrong with your root python?

If you `pip`, your system will `quit`

We have no way of ensuring changes for one project don't break another, like incompatible numpy requirements, or newly introduced bugs. Additionally, we have no way of replicating your environment for someone else's use.

Virtual envs

Cached, immutable packages are injected into an environment folder



Each env captures only what is necessary.

Ideally, env spec is frozen, and never altered needlessly.

Or, it is rebuilt from (changed) code or specs

Environments, Apps, Libraries

> *"There is a subtle but very important distinction to be made between **applications** and **libraries**. This is a very common source of confusion in the Python community."*
>
> *— Kenneth Reitz, pipenv*

Environments, Apps, Libraries

Root Environment:

A 'Matlab' or 'R'-like ***global*** set of tools, which you hope will all work together seamlessly (hint: they won't).

Application Environment:

A ***specific, exhaustive*** set of dependencies to make one project work repeatably, intended for ***deployment***.

Library "Environment":

Designed to be deployed into *another* environment and play nicely, it specifies the ***minimal*** set of deps, eg `numpy>=1.0`

Library Development App:

A special app used just to develop your library

Environments, Apps, Libraries

An advanced Data Scientist should use:

Root Env

Infrequently

- Stuff to manage your apps (eg, cookiecutter)
- Stuff with *no* dependencies

Apps

Most of the time

- Specify or freeze their deps

Libraries

Things you reuse between projects

- Test inside your app envs

Virtual Environments

The first step in repeatability is building an isolated env from scratch

For the Purist:

```
python -m venv ENV
source ENV/bin/activate
pip install -r requirements.txt
```

- May include system packages for binaries

For the Scientist:

```
conda env create -f env.yml
```

- Need to juggle conda vs pip

New For the Bold:

```
pipenv install
```

- Based on venv

Ensure env spec is defined in code. Never `pip install` a package manually, instead update `requirements.txt` and rebuild

What are your reqs?

Library

Eg, what's in `setup.py`

`requirements-to-freeze.txt`:

```
requests[security]
flask
gunicorn==19.4.*
```

```
(dev) pip install -r requirements-to-freeze.txt
(dev) pip freeze > requirements.txt
```

Frozen App

Eg, what's in your `Makefile`

`requirements.txt`:

```
cffi==1.5.2
cryptography==1.2.2
enum34==1.1.2
Flask==0.10.1
gunicorn==19.4.5
idna==2.0
ipaddress==1.0.16
itsdangerous==0.24
Jinja2==2.8
...
```

```
(deploy) pip install -r requirements.txt
```

Pipenv

```
pipenv install numpy
```

Minimal Pipfile:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[packages]
numpy = "*"

[dev-packages]
pytest = "*"
```

Frozen Pipfile.lock:

```
{
    "default": {
        "numpy": {
            "hashes": ["sha256:40523d..."],
            "version": "==1.15.0"
        }
    },
    "develop": {
        "pytest": {
            "hashes": ["sha256:f46e49..."],
            "version": "==3.2.2"
        }
    }
}
```

Conda

(of the Anaconda python distribution)

Pip + compiled packages

Great for science - historically better numpy/scipy/pandas installations

Specify deep stuff like OpenBLAS, Intel MKL, …

With its own virtualenv-like conda environments

Conda

The Good

```
name: myenv
dependencies:
  - numpy
  - pip:
      - django
```

Flexible, powerful, more specific than requirements.txt

The Bad

```
$ conda list -e
# platform: osx-64
appnope=0.1.0=py36_0
numpy=1.14.2=py36ha9ae307_1
...
```

Builds and even packages may be architecture specific, eg appnope

See also: `conda env export --no-builds`

The Ugly

```
name: myenv
dependencies:
  - tornado>=4.3
  - pip:
      - flower==0.9.1
```

Flower incorrectly requires tornado v4.2.0, ***removing*** the conda package and replacing it via pip

TL;DR

Truly repeatable deployment for your project is not a solved problem for python

Each method has pros and cons, and you need to be vigilant

This class will explore pipenv and conda, making up the slack with docker.

My Recommendations

Use Pipenv

- For pure-python projects
- … or basic pydata stack
- … or with Docker if you don't mind slower rebuilds, or have a robust container registry

Use Conda

- For compiled tools
  - … or non-python deps, eg
    - GDAL, OpenBlas…
- With semi-frozen `env.yml`
  - or 1 env spec/platform

Lore

Related: Instacart released a tool called lore which they claim solves python dependecy issues.

It rejects pipenv for being too new (so they release something newer?)

It poses as a lean environment tool but is really a full-stack machine-learning package - which is not required for most projects

Why not just contribute to pipenv???

I think this is a failure of isolation and a failure of collaboration.

Testing

What to test

The environment

You can call your deps!

Your deps work correctly (eg, numpy/blas)

Your libraries

In a matrix and ***in your app env***

You probably haven't tested them as well as other libs

Your app

This should be obvious

And coverage

How much is tested?

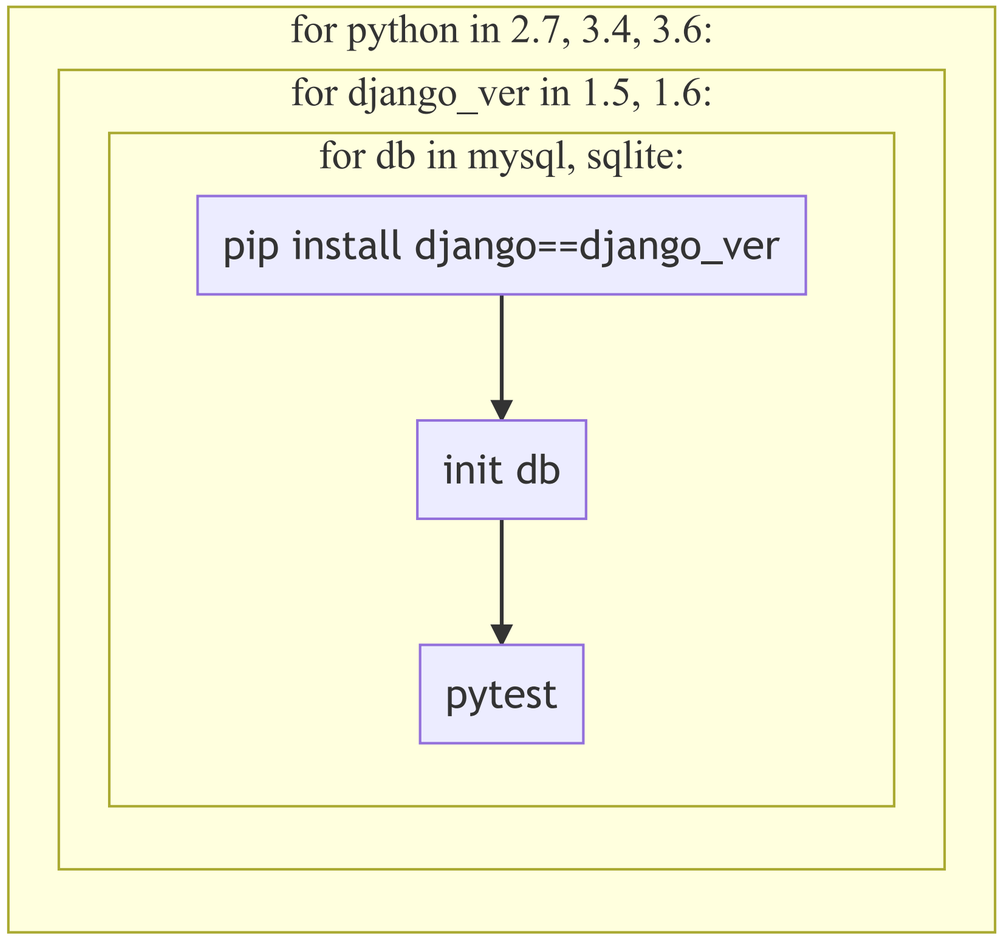
Your integration

Ideally, continuously deploy your ***actual executables*** to a staging environment

Test Matrix

Eg, tox:

```
[tox]
envlist = py{27,34,36}-django{15,16}-{sqlite,mysql}
```



Two faces of testing

Suites

```
from unittest import TestCase

class TestStringMethods(TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
```

... and discovery

```
python -m unittest my_library
python setup.py test
pytest
```

Test Suites

```
from unittest import TestCase

class TestStringMethods(TestCase):

    def setUp(self):
        # Common code run before every test
        pass

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
```

Unittest is built in

See also Software Carpentry: CI and Testing

Pytest

Pytest is an emerging standard for testing in python.

It provides…

Pytest

less boilerplate

```
# content of test_sample.py
def inc(x):
    return x + 1

def test_answer():
    assert inc(3) == 5
```

Pytest

and better test runners

... with plugins!

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_sample.py F                                                  [100%]

================================ FAILURES ================================
```

```
_____________________________ test_answer _____________________________

    def test_answer():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_sample.py:6: AssertionError
======================== 1 failed in 0.12 seconds ========================
```

Pytest

However...

Consider whether you need pytest's advanced authoring tools

Most tests can ***and should*** be written with vanilla `TestCase`'s

Try to use pytest only as a ***test runner***

You may need to switch runners later, and don't want to re-write your tests!

Pytest

Avoid magic…

While powerful, fixtures exhibit ***bad implicit design***

```python
@pytest.fixture
def smtp_connection():
    return SMTP("smtp.gmail.com", 587, timeout=5)


# "Magic" arg injection
def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    ...
```

Pytest

Avoid magic…

… you don’t always need it!

```
from functools import wraps

def smtp_connection(func):
    @wraps(func)
    def wrapped(*args, **kwargs):
        kwargs['smtp_connection'] = SMTP()
        return func(*args, **kwargs)
    return wrapped

class SmtpTests(TestCase):
    @smtp_connection
    def test_ehlo(self, smtp_connection=None):
        response, msg = smtp_connection.ehlo()
        ...
```

Coverage

What you have tested!

```
$ ./drun_app pytest
============================= test session starts ==============================
platform linux -- Python 3.6.6, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /app, inifile: setup.cfg
plugins: cov-2.6.0
collected 2 items

pset_utils_gorlins/tests.py .                                            [ 50%]
tests/test_pset_utils_gorlins.py .                                       [100%]

----------- coverage: platform linux, python 3.6.6-final-0 ---
Name                                             Stmts   Miss  Cover
--------------------------------------------------------------------
pset_utils_gorlins/__init__.py                      10      0   100%
pset_utils_gorlins/pset_utils_gorlins.py             5      0   100%
--------------------------------------------------------------------
TOTAL                                               15      0   100%


========================== 2 passed in 0.39 seconds ============================
```

Coverage

```
def f(x):          # line 1
    if x:          #      2
        y = 10     #      3
    return y       #      4

def test_f():
    assert f(5) == 10
```

Line Coverage

100%

Branch Coverage

`f(0)` will raise error!

Missing ‘branch’ condition where `if` evaluates `False`, ie no coverage between lines 2->4

Test structure

Standard

```
your_project/
    __init__.py
tests/
    test_your_project.py
setup.py
```

```
from your_project import some_function
def test_some_function():
    ...
```

But I recommend...

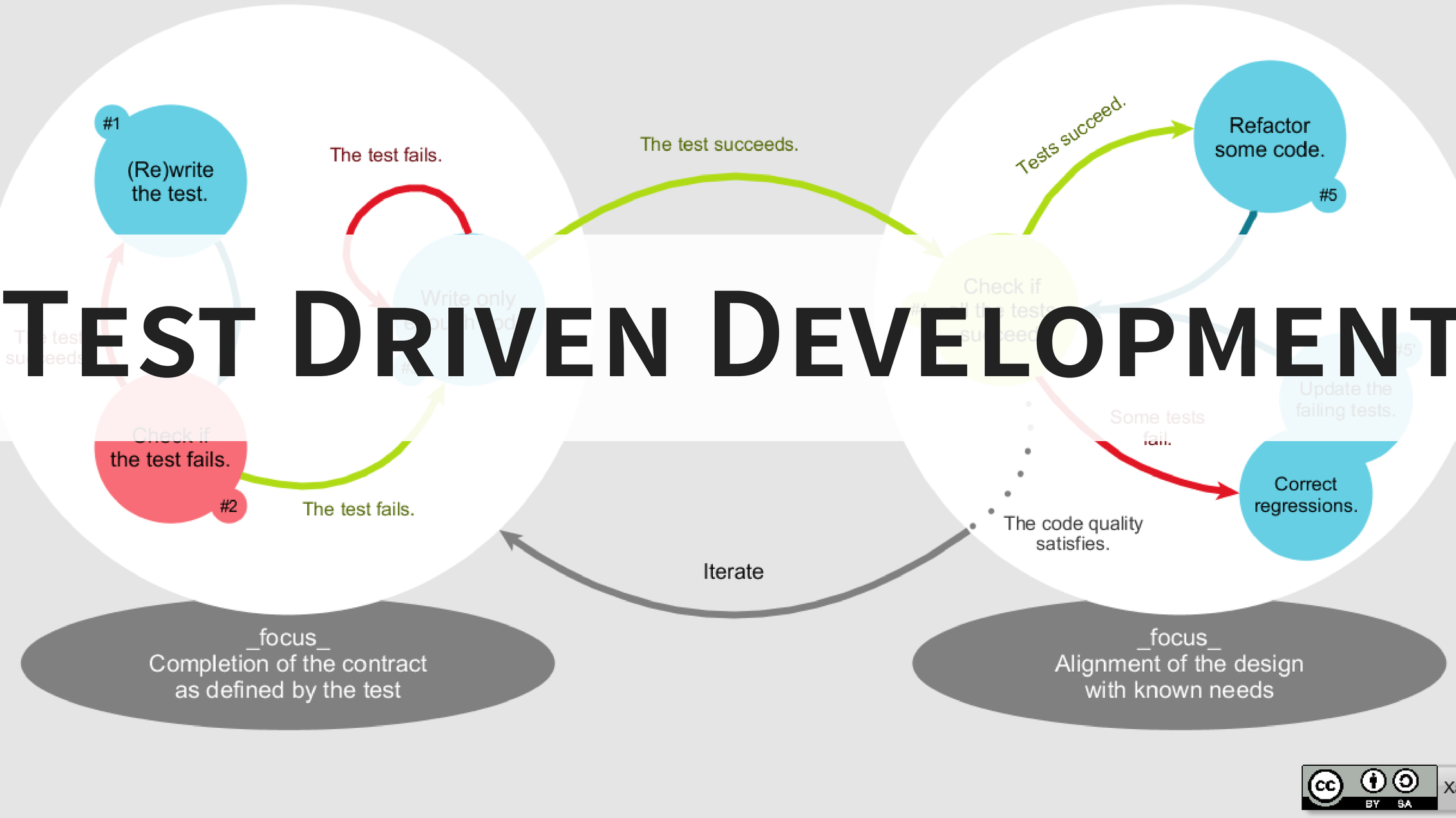
```
your_project/
    __init__.py
    tests/
        __init__.py
        test_aspect.py
setup.py
```

```
from .. import some_function
def test_some_function():
    ...
```

… because it is testable on deploy!

```
pip install your_project
pytest your_project
```

Test Driven Development



Test Driven Development

Test

```
def increment(x):
        raise NotImplementedError()
```

```
class IncrementTests(TestCase):
    def test_increment(self):
        for a in [1, 2, 3]:
            assert increment(a) > a
```

Develop

```
def increment(x):
    return x + 1
```

Focus on the function interface and expected return

Testing First vs Code Debt

Tyrion never wrote a unit test.

You will never repay yours!

You will never convince your boss to let you go back and test!

Testing is part of the ***definition of done***.

Testing tools

Hypothesis

Hypothesis helps to generate test cases

NB: I have not used personally

```
from hypothesis import given
from hypothesis.strategies import text

class TestEncoding(TestCase):
    @given(text())
    def test_decode_inverts_encode(self, s):
        self.assertEqual(decode(encode(s)), s)
```

Mock

Mock allows you to replace parts of your system under test with mock objects and make assertions about how they have been used

It allows you to ***duck-punch*** out objects so you can test without so many side effects, or isolate code that is not easy to unit test

```
class ProductionClass:
    def method(self):
        self.something(1, 2, 3)
    def something(self, a, b, c):
        ...

def test_something():
    real = ProductionClass()
    real.something = MagicMock()
    real.method()
    real.something.assert_called_once_with(1, 2, 3)
```

Mock

However...

Mock is easy to abuse, by testing the *implementation* more than the *result*.

Do you care if `.something()` was called? Or is this an ***implementation detail***?

If your tests assume too much about the implementation, you will have ***false positive test failures*** when the implementation changes.

Used correctly, `mock` is one of the most powerful tools you have. Be sure not to overly complicate things!

Mock

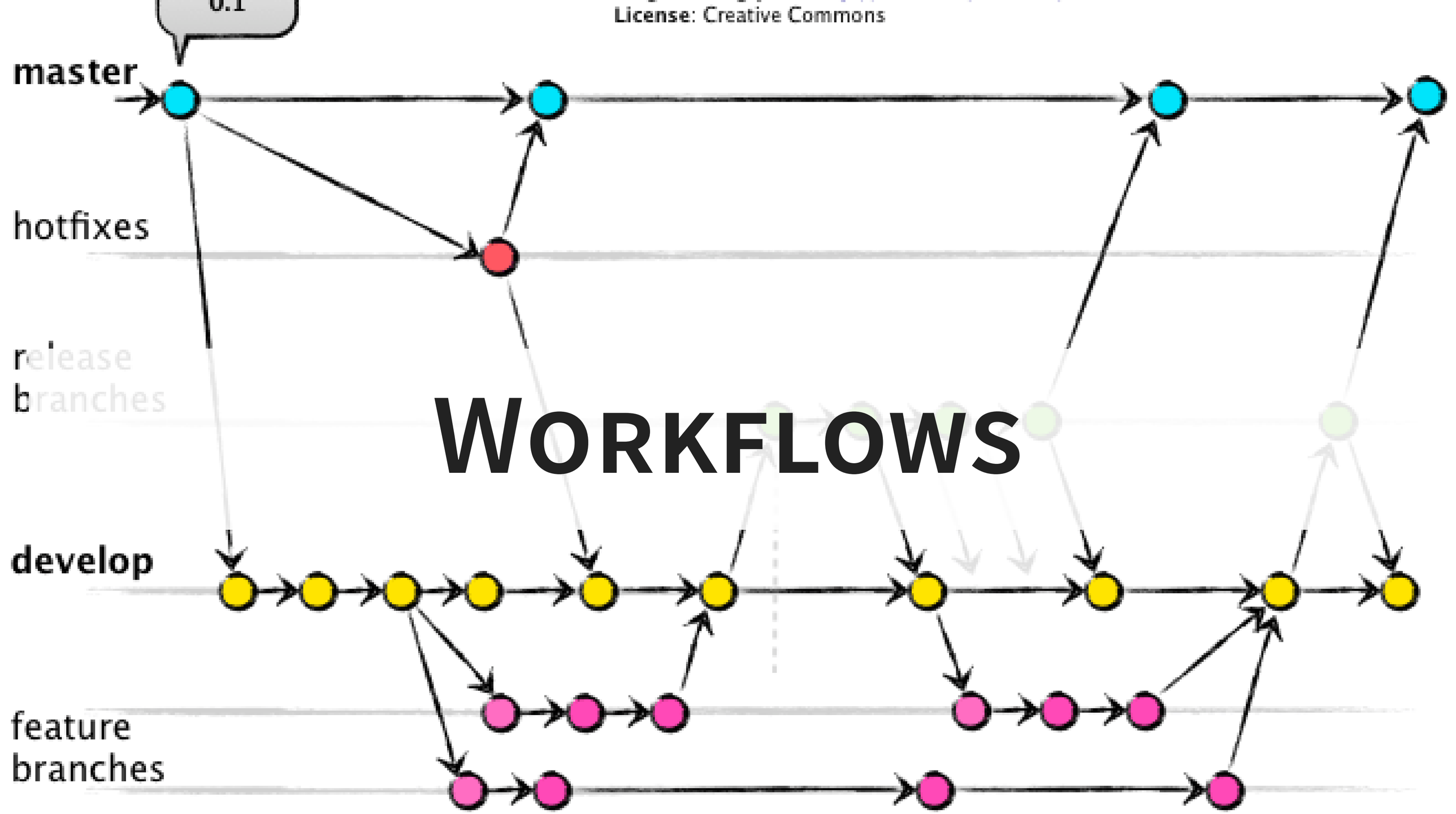
Recall - testing should be ***stateless*** and free of side-effects. Your tests should not change your prod data!

Therefore, the real value of mock is providing testable stubs for “integration” testing, eg…

Moto

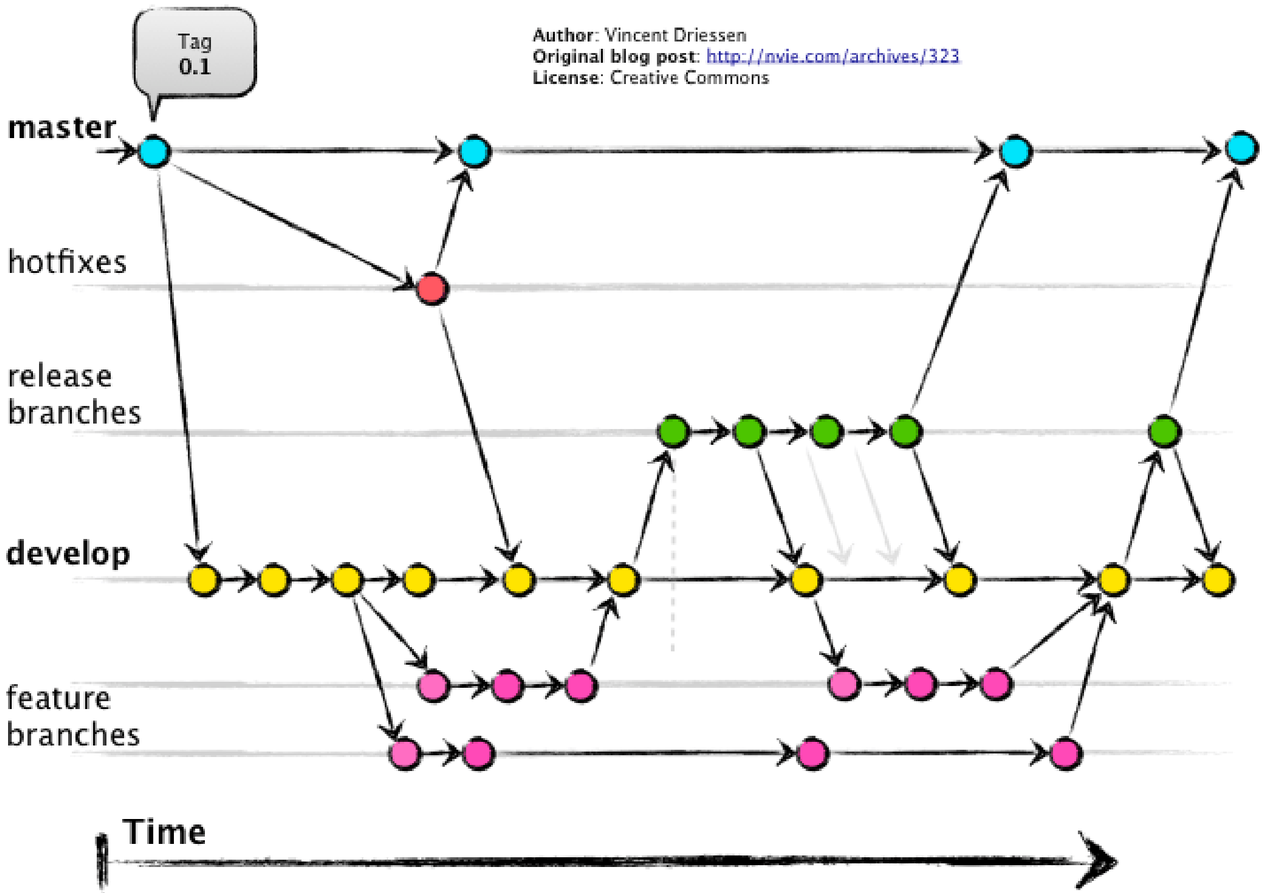
Moto is Mock for Boto (AWS client)

```
import boto
from moto import mock_s3
from mymodule import MyModel

@mock_s3
def test_my_model_save():
    conn = boto.connect_s3()
    # We need to create the bucket since this is all in Moto's 'virtual' AWS account
    conn.create_bucket('harvard')

    model_instance = MyModel('scott', 'is awesome')
    model_instance.save()

    assert conn.get_bucket('harvard').get_key('scott').get_contents_as_string() == 'is awesome'
```


0.1
License: Creative Commons
master
hotfixes
release
branches
develop
feature
branches
Time

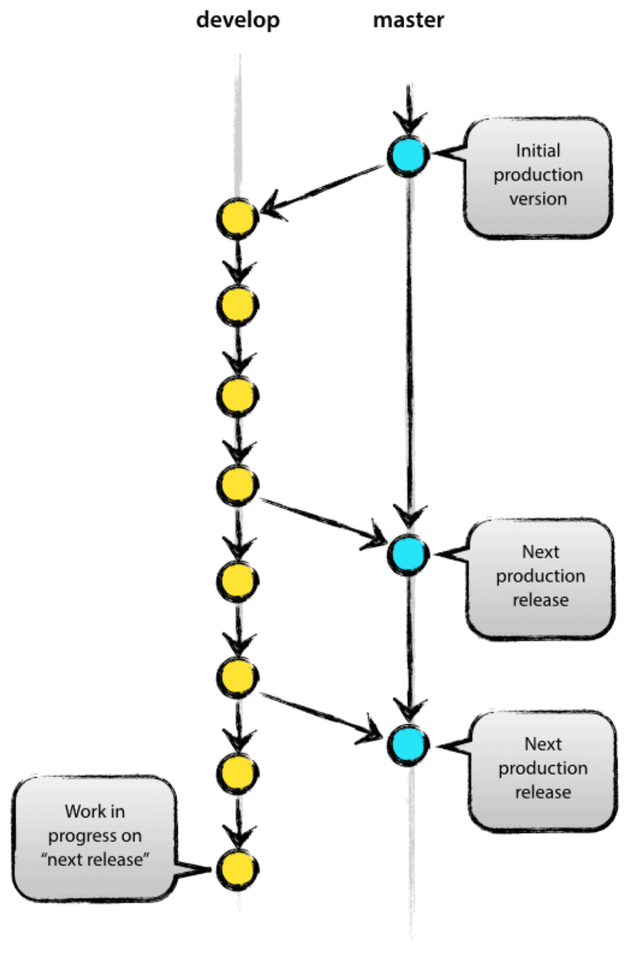
Workflows

Git Flow

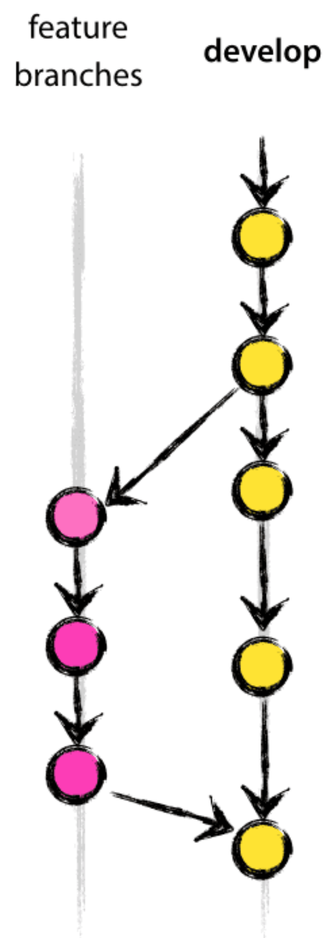


A Successful Git Branching Model

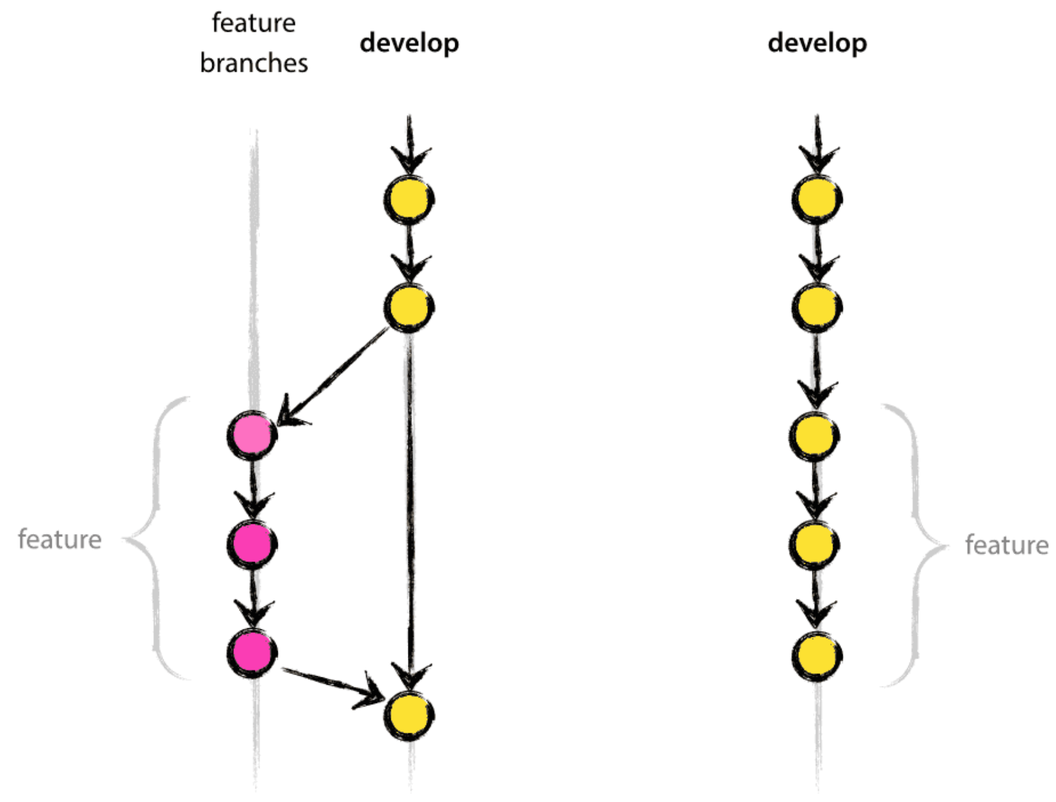
Releasing


develop
master
Initial production version
Next production release
Next production release
Work in progress on “next release”

Feature Branches



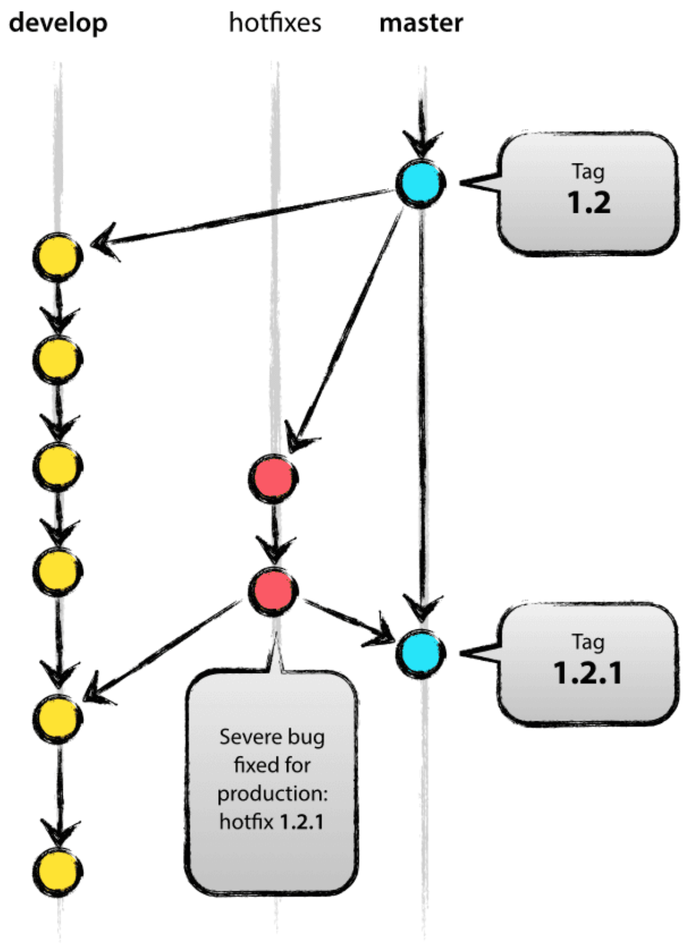
Merging



`git merge --no-ff`

`git merge`
(plain)

Hotfix


develop
hotfixes
master
Tag
1.2
Tag
1.2.1
Severe bug
fixed for
production:
hotfix 1.2.1

Bootstraping

We don't want to start from scratch

Cookiecutter is a templating system and a repository for best practices

It will kick us off (and standardize our future work)

Discussion Time

Cluster at your table - at least 1-2 computers w/ wifi, 4+ students

https://cookiecutter.readthedocs.io/

Browse through ‘Available Cookiecutters’

Focus: `cookiecutter-pyproject` and `cookiecutter-data-science`

Discussion Time

Explore with your group, identify a few pros and cons about each:

cookiecutter-	Pros	Cons
pypackage	• Testing: Travis, Tox • setup.py • pytest	• requirements_dev.txt • bumpversion • click
data-science	• Sync to S3 • Makefile • Opinions in docs	• No conda • Data distinctions • References, non-VCS stuff

Declarative State

Declarative vs Procedural

Note how we are changing focus from….

Procedural

Telling the computer what to do

```
pip install numpy
```

```
# Write 'world' to 'hello.txt'
with open('hello.txt', 'w') as f:
    f.write('world')
```

… to Declarative

Describing the final state

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

```
def ensure_content(file, content):
    fp = '{}'.format(
        sha256(content.encode()
        ).hexdigest())
    if not os.path.isfile(fp):
        with open(fp, 'w') as f:
            f.write(content)
    os.link(fp, file)
```

Keep this is mind as a theme/pattern for higher level coding

PSET 1

Cookiecutter

With weekly psets, you'll have a lot of repeat work

How better to help you out than create a project template for this class?

The Utils library

No one likes unnecessary boilerplate

We'll create a utilities library for use throughout the class

You don't need to package ***every single thing you do*** separately

TravisCI

We'll get live CI builds on your repos thanks to TravisCI and their educational deals with GitHub.

Your work will be tested, making it easier to develop, and easier for us to evaluate.

Atomic Writes

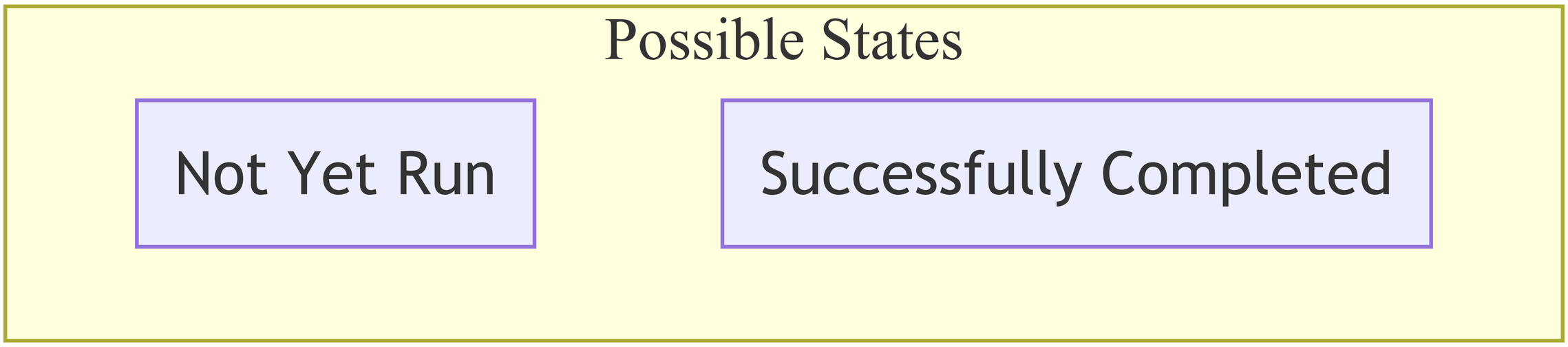
… ensuring your files are full and complete and (ideally) idempotent

```
# This the BAD way
big-slow-calculation > /outputs/foo.data
```

```
# This is the good way
big-slow-calculation > /outputs/foo-tmp-123456.data
mv /outputs/foo-tmp-123456.data /outputs/
```

Atomic Writes

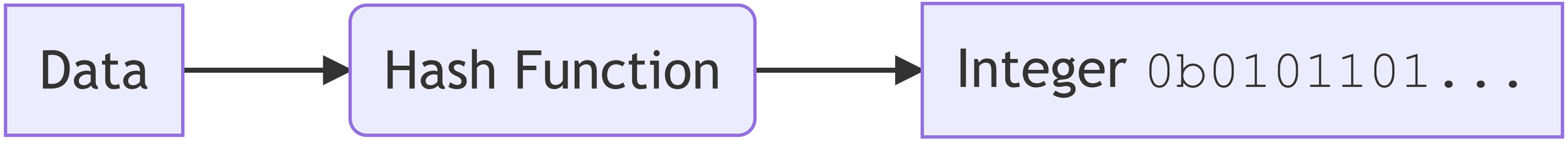
The system can never be in between:



Hashing

Prepping to explore some pseudo-sensitive data (eg, you!)

Hashing



A hash function, like sha256, maps an arbitrary string of data into a fixed-width integer, eg 256 bits

The integer appears ***uniform*** and is deterministic, which makes it very useful for ***anonymity***, without a DB tracking name -> id

Readings

- Luigi Docs
- Efficient Estimation of Word Representations in Vector Space
- Gensim Word2Vec
- Word Embeddings: A Natural Language Processing Crash Course