Workflows

Keeping Things Together

Dr. Scott Gorlin

Harvard University

Fall 2018

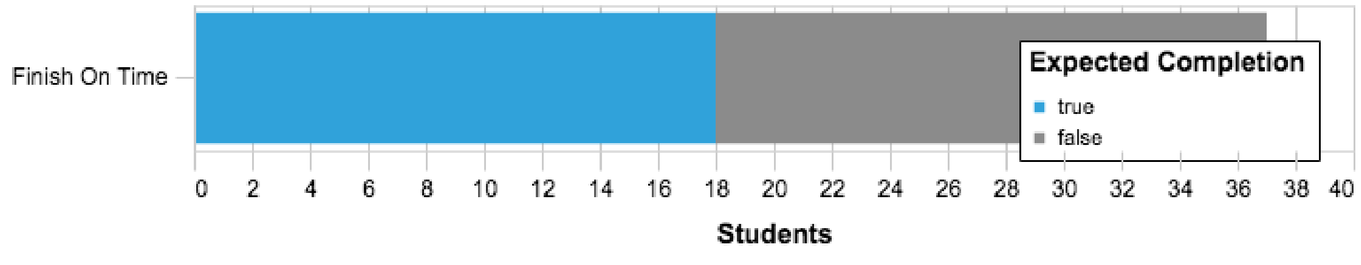
Agenda

- Pset Utils
- Debugging
- Branch Workflows
- Versioning
- Higher Levels
- Config
- Word Embeddings
- Readings

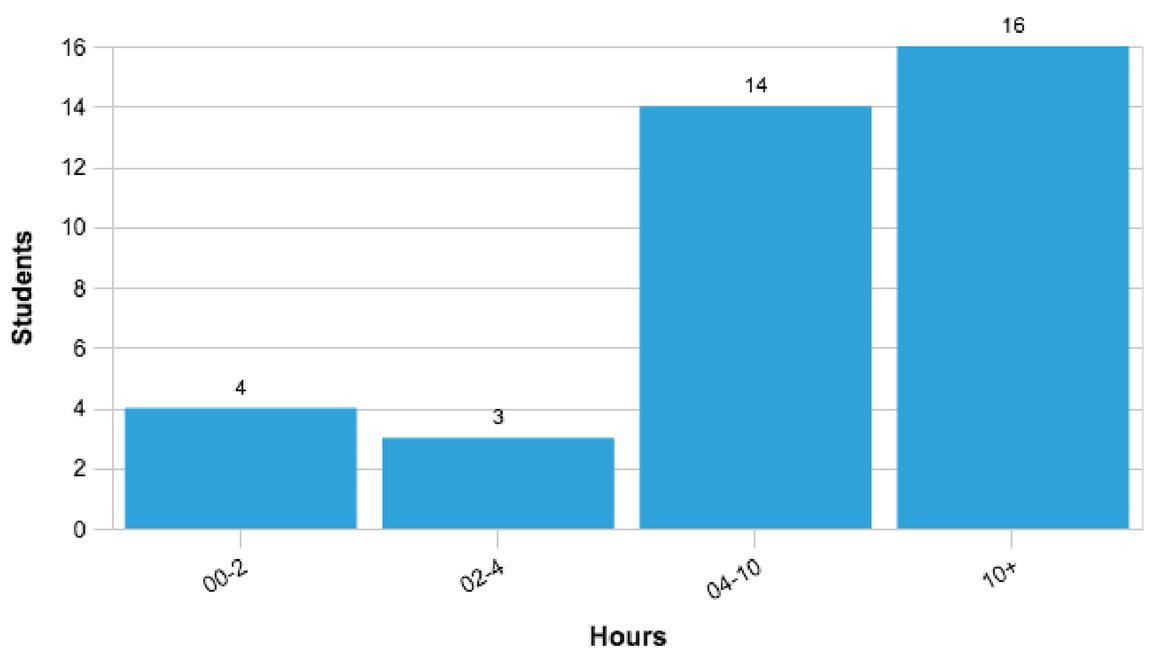
Pset Utils

Pulse

Expected Complete On Time



Total Hours on Pset Utils



Piazza Hall of Records

Top Contributors:

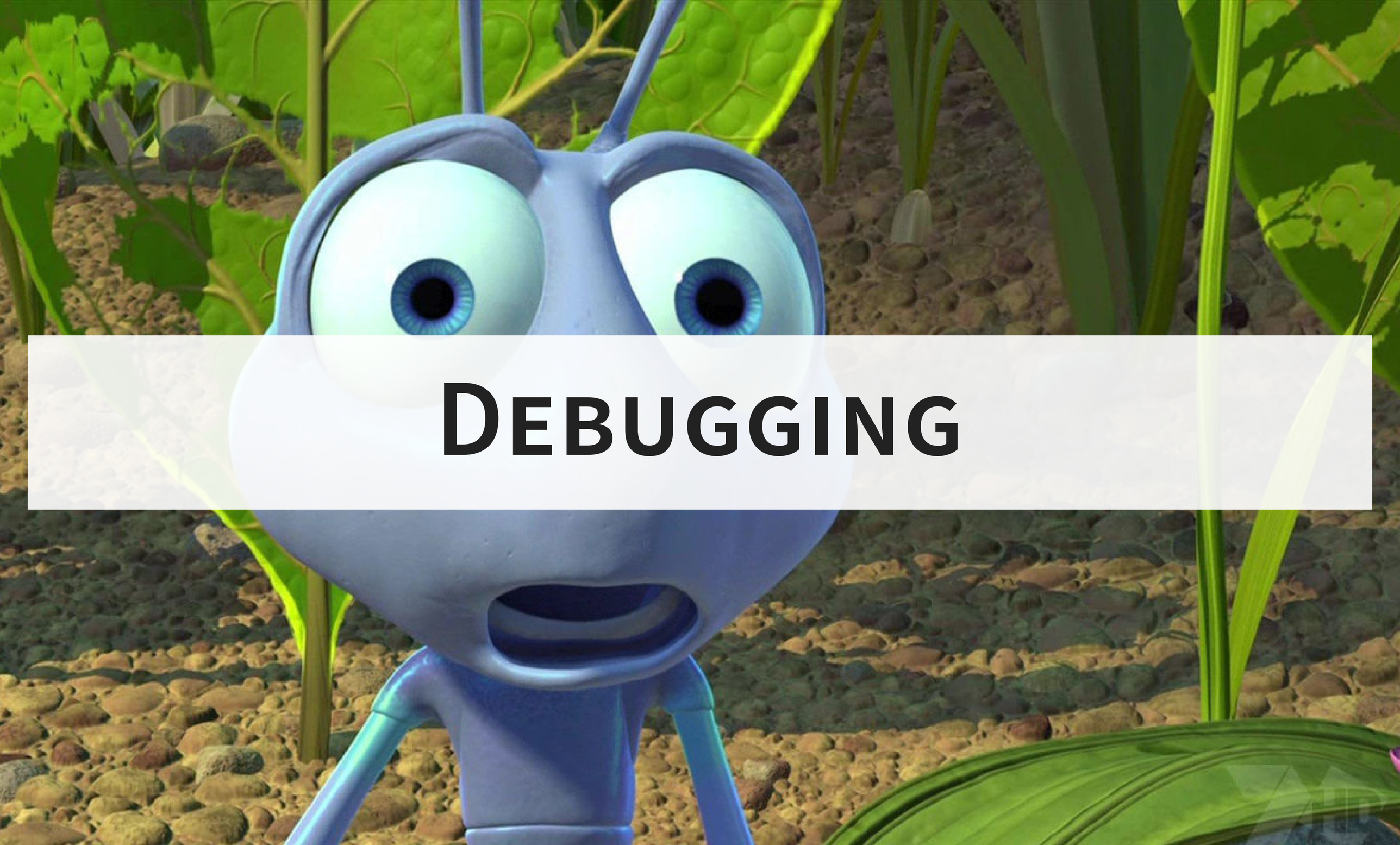
- 30 Aaron White
- 21 Jeff Winchell
- 19 Michael Blanchard
- 13 Amyrah Arroyo
- 13 Manish Sinha

Objectives

- To learn a templating system, why it's useful, and how to modify it
  - Both the project structure (cookiecutter) and the language (Jinja2)
- To explore a best-practices repository for reference later
- To save you time in this course, and give you tools to replace boilerplate in your real life
  - … and to reinforce the usage throughout the course
- To expose you to python packaging and testing
- To expose you to learning directly from documentation and, more likely, raw code

Next Pset Objectives

- Continue to modify and improve project template
- Practic maintaining your library
- Practice applying the library concept to new apps



Debugging

How to ask for help

This is not a ding based on last pset issues :). It is central to your development as a Data Scientist

Simply posting something like “I cannot get *X* to work” is not sufficient

You need to provide ***context*** and ***repeatability*** for your issues as you would for your code.

If you accurately understand and describe your problem, 80% of the time you will have answered it as well!

How to ask for help

"My tests aren't working" vs:

What did you do?

```
./drun_app pytest
```

What did you expect?

Tests to run (not necessarily pass)

What happened?

```
usage: pytest [options] [file_or_dir] [file_or_dir] [...]
pytest: error: unrecognized arguments: --cov=python_boilerplate --cov-branch
```

Install `pytest-cov` and GTG

How to ask for help

5 Why's

I have an error running tests **Why?**

- Unable to detect version **Why?**
- Not an intact git repo **Why?**
- Never `git init`'d my rendered template

Stack Trace

```
ERROR collecting tests/test_python_boilerplate.py
LookupError: setuptools-scm was unable to detect
version for '/app'.
Make sure you are either building from a fully
intact git repository or PyPI tarballs. Most other
sources (such as GitHub tarballs, a git checkout
without the .git folder) do not contain the
necessary metadata and will not work.
```

Debugging

If you find yourself randomly trying various permutations in hopes that it will “just work,” stop and try to understand the problem

Ask for help when you can’t make progress answering a ‘Why?’ or aren’t learning about how the thing works.

Have fun and don’t get frustrated 🙂

Logging Info

Print

`print` is for output, `Logger`'s are for debug

Only use `print` if your app is designed to output an answer to stdout

For all other usages, `print` is tempting but ultimately limiting

Loggers

If you need info for debugging your program, use these to manage it

But, they are a poor replacement for step-through debugging. Use for info from processes or hard-to-debug internals etc.

Logging Info

Print

`print` is for output, Logger's are for debug

Only use `print` if your app is designed to output an answer to stdout

For all other usages, `print` is tempting but ultimately limiting

Loggers

```
from logging import (
    root, INFO, StreamHandler,
)

logger = root


# By default you won't see everything
logger.setLevel(INFO)

# Tell it where to log
logger.addHandler(StreamHandler())
```

```
logger.info("Hello!")
# INFO:root:Hello!
logger.warning("Beware!")
# WARNING:root:Beware!
```

Logging Info

Using loggers allows you to:

- Isolate info to levels: `FATAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`, …
  - Can easily tweak verbosity of your app
- Set the logging format, eg:
  - `INFO:root:Hello!`
  - `2018-09-18 12:58:04,853 INFO Hello!`
- Handle multiple destinations: stdout, a log file, …
- Handle multiple *loggers* - your app, a system function, …

Which logger?

Root

```
logger = logging.root
```

Your App

```
logger = logging.getLogger(
    __name__
)
...
from your_app import logger
```

A framework's

```
logger = logging.getLogger(
    'luigi-interface'
)
```

Use the object!

```
def func(logger=None):
    if logger:
        logging.info('Hello!')
    return another_func(logger=logger)

func(logger=my_logger)
```

Practical Considerations

Logging still occurs even if it's suppressed…

Avoid:

```
logger.info(slow_calculation())
```

… because it still runs even if the output is suppressed

If you must…

```
def lazy_info(callable, logger=None):
    if logger and logger.level <= logging.INFO:
        logger.info(callable())

lazy_info(
    lambda : get_accuracy(model, x, y),
    logger=logger)
```

0.1

master

hotfixes

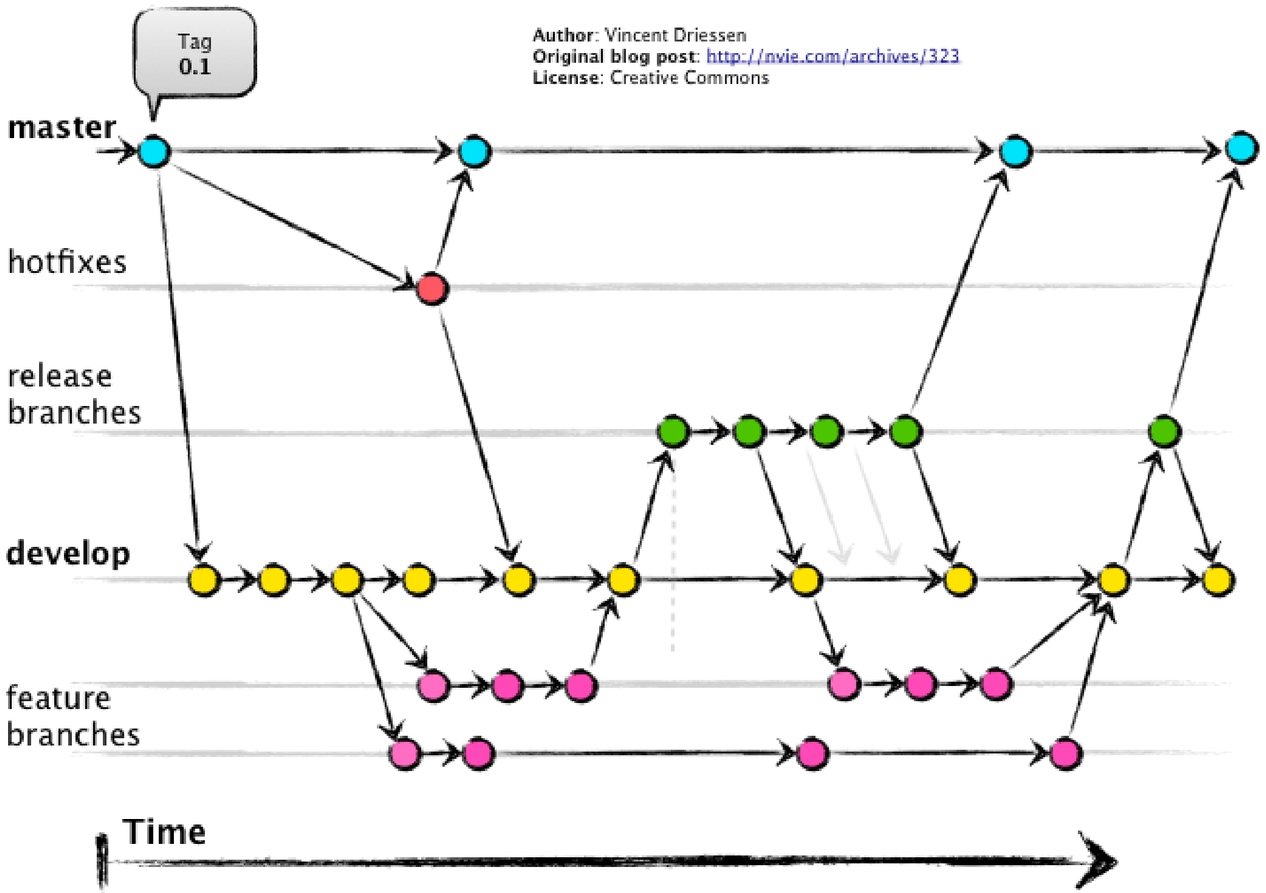
release branches

develop

feature branches

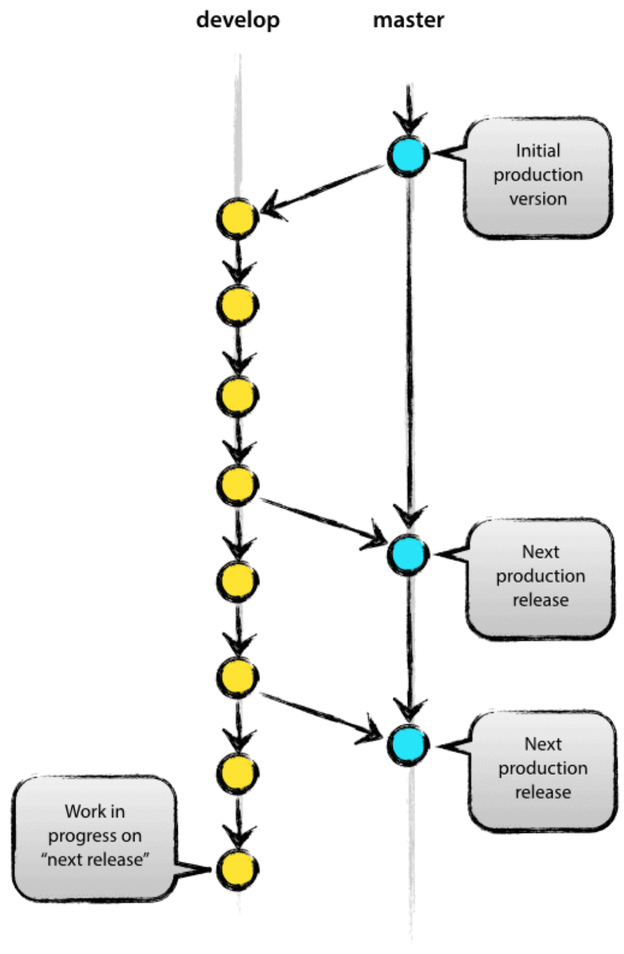
Branch Workflows

Git Flow

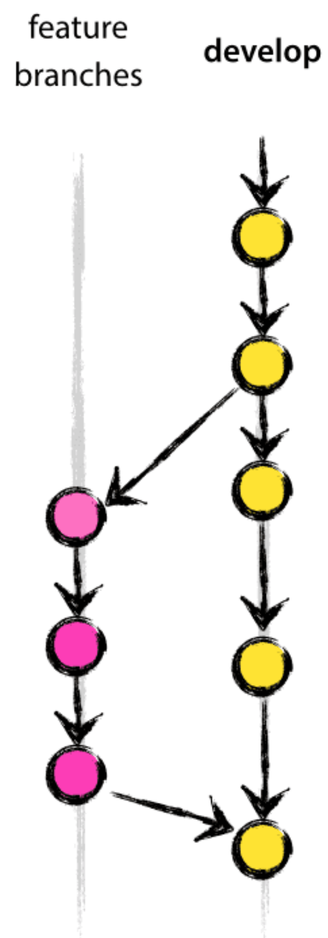


A Successful Git Branching Model

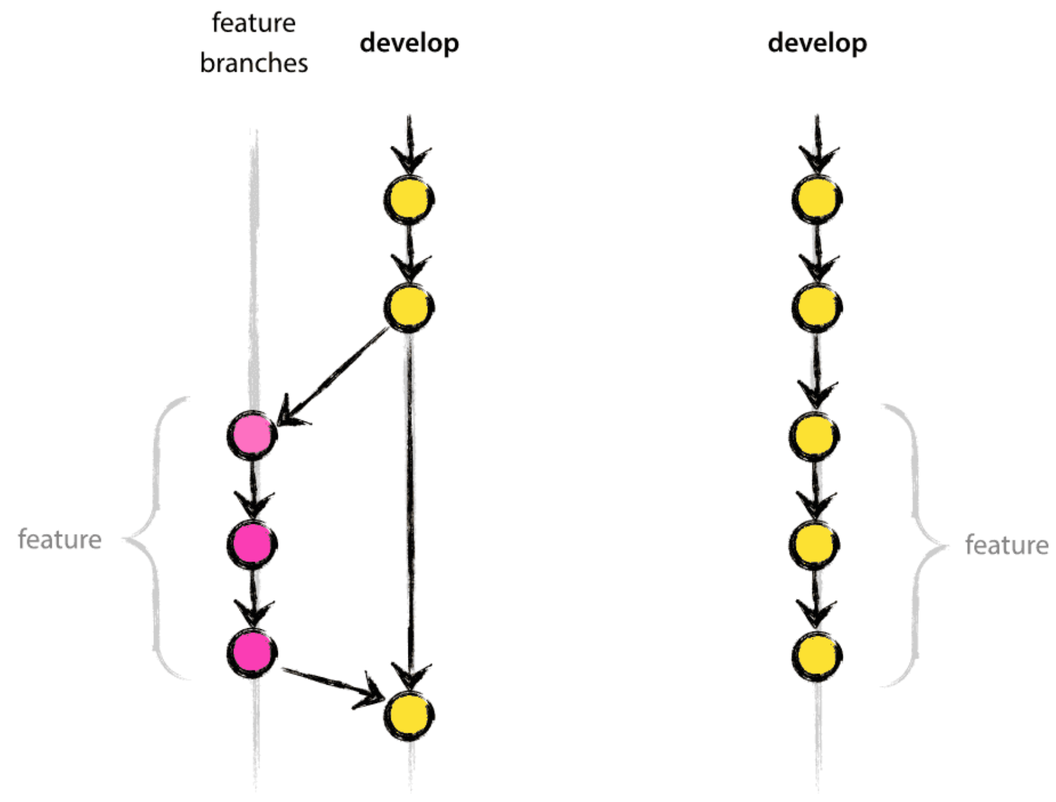
Releasing


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

Feature Branches



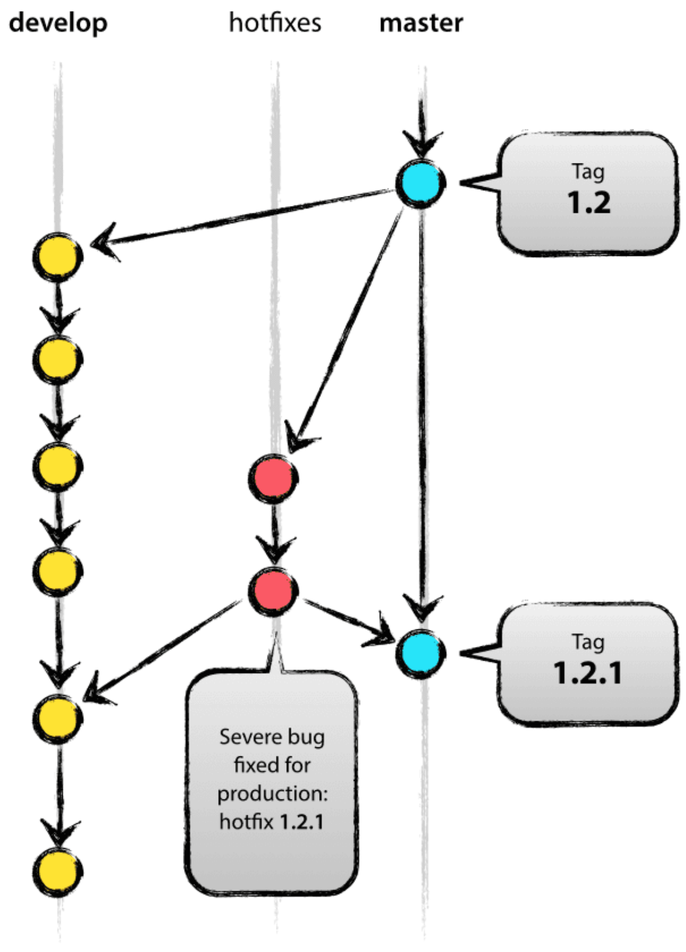
Merging



`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

Versioning

Semver

Version numbers are meaningful!

Given a version number `{major}.{minor}.{patch}`, increment:

- **Major** when you make backwards-incompatible changes,
- **Minor** when you add backwards-compatible features, and
- **Patch** when you fix bugs.

Requirements w.r.t Semver

- Should be safe to require x . y . *
- Specify minor version you ***need***, e.g. >=1 . 2
- Specify major version, e.g. =3 . *
  - Or, live a little and hope for the best. Apps are frozen, and if you don't ***know*** there's an issue in the next version, it's okay not to protect against it

Tracking the version

Most packages specify their versions ***twice***!

The Install

```
# What's set in setup.py
import pkg_resources
pkg_resources.get_distribution("scipy").version

# site-packages/
# └── scipy-1.1.0-py3.6.egg-info/
#     └── PKG-INFO
```

The Code

```
import scipy
print(scipy.__version__)  # By convention

# site-packages/
# └── scipy/
#     ├── __init__.py
#     └── version.py
```

Hopefully they agree!

Keeping your version consistent

DRY: Don't Repeat Yourself

Need to find a way to specify version ***once*** and get it right

Need version to be globally ***unique***

Need to account for dev versions for your own code. We may test, read docs, install (accidentally or by necessity), etc from any random commit with much higher likelihood than any other package

Keeping your version consistent

Simple: specify via hardcode in `version.py` and import in `setup.py`.

But:

- Sometimes import issues
  - Importing from your code before `setup_requires` packages are installed is risky!
- Prone to forgetting to commit version number changes
- Doesn’t specify commit for dev versions

Bumpversion

Some packages automate version increments, e.g. Bumpversion and bump

They basically grep your `version.py` and automatically rewrite it

These automate parts of the process but have other issues - you still must bump manually on release, sometimes grep fails, and they don't distinguish code during development

Why not just git?

Your git history contains all you need to know to understand the code

(right?!?)

Your git commit ID ***is*** a unique ID that can specify your version exactly.

However, it is unintuitive and non-sequential

Is `6e50e7` more recent than `b79cb6`?

Can we automate versioning via git?

Why not just git?

Some tools do:

- Semantic Release
  - Cf: Original for Node.js
  - Inspects git commit comments in Angular format to determine if release is patch, minor, or major
- Python Build Reasonableness similarly will inspect git commits

> But, git history should be ***nice to have***, not ***necessary*** - having to 'debug' your commit history to get a correct release version feels backwards.

SETUPTOOLS_SCM

the blessed package to manage your versions by scm tags

```
# setup.py
from setuptools import setup
setup(
    ...,
    use_scm_version=True,
    setup_requires=
['setuptools_scm'],
    ...,
)
```

```
# your_package.__init__
from pkg_resources import get_distribution, DistributionNotFound
try:
    __version__ = get_distribution(__name__).version
except DistributionNotFound:
    # package is not installed
    from setuptools_scm import get_version
    __version__ = get_version(root='..', relative_to=__file__)
```

If code is Semver tagged, `{major}.{minor}.{patch}`

Otherwise, `{next_version}.dev{distance}+{scm letter}{revision hash}`

`0.2.2.dev0+gd89ddd2.d20180915`

Logging

We now have a completely unique, well-formed version, specified once

Use for reqs, auditing, etc

```
from logging import root
import my_app

root.info("Running {} v{}".format(my_app.__name__, my_app.__version__))
```

Versioneer?

A challenger? Or a distraction?

Be wary of non-standard, not-invented-here tools

But embrace them if they’re better!

Higher Levels

Decorators

We've seen this recently:

```
@some_decorator  # Note lack of ()
def some_func():
    ...
```

Which is syntactic sugar for…

```
# Note it decorates a single arg
def some_decorator(func):
    ...

def some_func():
    ...

some_func = some_decorator(some_func)
```

A decorator intercepts a function or class after it's defined. It can wrap the function, alter it, register it, or remove it.

Decorators are awesome, but easily the most confusing syntax in all of Python

Registries

We don’t have to modify a function - we can register it

```
SUMMARIES = {}

def register(func, name=None):
    SUMMARIES[name or func.__name__] = func
    return func

@register
def mean(x):
    ...

def summarize(vec, stat='mean'):
    return SUMMARIES[stat](vec)
```

Wrappers

When you wrap a function inside another.

You can easily make the wrapped function look like the original (ie preserve docs, signatures, etc) using `functools.wraps`.

```
from functools import wraps
from logging import root

def logme(func):
    # Create a new function
    @wraps(func)
    def wrapped(*args, **kwargs):
        root.info(
            "{} was called: {} {}".format(
                func.__name__, args, kwargs))
        # Encapsulate the original
        return func(*args, **kwargs)
    return wrapped  # Return the new wrapper
```

```
@logme
def x(*args, **kwargs):
    pass

x(1, a=24)
# x was called: (1,) {'a': 24}
```

Decorators – With Args

Beware the ( )!

Vanilla

```
@decorator
def func(x):
    ...

# Equivalent to
func = decorator(func)
```

Parameterized

```
@decorator()
def func(x):
    ...

# Equivalent to
func = decorator()(func)
```

Decorators – With Args

Parameterized decorators must encapsulate two functions!

```
def logme(logger=root):
    def decorator(func):
        @wraps(func)
        def wrapped(*args, **kwargs):
            ...
        return wrapped
    return decorator

@logme()
def f(x):
    ...
```

Decorators – With Args

… or juggle the kwargs to allow both:

```
@logme
def f(x):
    ...

@logme(logger=my_logger)
def g(x):
    ...
```

```
def logme(func=None, logger=root):
    if func is None:
        return (
            lambda func: logme(
                func=func, logger=logger
            )
        )
    @wraps(func)
    def wrapped(*args, **kwargs):
        ...
    return wrapped
```

Context Managers

One form of ‘Higher Level Code’ is the context manager:

> *A context manager is an object that defines the runtime context to be established when executing a with statement*

Context Managers

Managers codify init-try-finally blocks around other code

```
class MyContext(object):

    def __enter__(self):
        ...
    def __exit__(self, exc_type, exc_val, exc_tb):
        ...

with MyContext() as context:
    ...
```

```
mgr = MyContext()
context = mgr.__enter__()
try:
    ...
finally:
    mgr.__exit__()
```

Context Managers

You've probably already used them:

```
class open(File):

    def __enter__(self):
        self.open()
        return self

    def __exit__(self, *exceptions):
        self.close()

with open('something.txt') as f:
    f.read()
```

Ctx Mgrs

Convenient shorthand: use the generator!

```python
from contextlib import contextmanager

@contextmanager
def write_signature(*args, **kwargs):
    with open(*args, **kwargs) as f:
        yield f
        f.write('\n\nSincerely,\n\n-Scott')

with write_signature('letter.txt', mode='w') as f:
    f.write("Hello!")
```

```
Hello!

Sincerely,

-Scott
```

Context

More info, see:

- Pep 343

Config

Your code is a 13 year old kid

… it can't keep a secret

> *A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.*
>
> — *The Twelve-Factor App*

Absolute Nothing

Your code should contain few, if any, absolute references - including to local disk systems, server URL's, etc. We need to configure these variables through the environment.

... EXCEPT FOR RELATIVES

You can use relative locations for code deploys - eg within the repo `./data`, which is git ignored. This isn't perfect - it doesn't work for libraries - but on deploy you can symlink these to other locations if necessary.

Config Basics

You can simply read any string value:

```
import os
secret = os.environ.get('MY_APP_SECRET') or ''
```

Or you may need to parse it if you want an int, bool, etc

```
# Choose either python or json syntax
from ast import literal_eval  # Never 'eval'

MY_FLAG = bool(    # Handle 0, False, 1, True, etc
    literal_eval(  # Or json.loads
        os.environ.get('MY_APP_FLAG') or '0'
    )
)
```

Config Basics

Not all configs are secrets. You can distinguish between ‘deploy configs’, eg pointing to servers and files, and ‘code configs’, eg running with verbose debugging, or higher training epochs, etc.

For the latter, it may make sense to version control the config (or set profiles in code, etc) since they are likely shareable between deploys.

Dotenvs

A popular technique is to use a `.env` file or `env` directory to specify many env variables at once. Pipenv and docker automatically read these, and other tools like envdir can load them explicitly

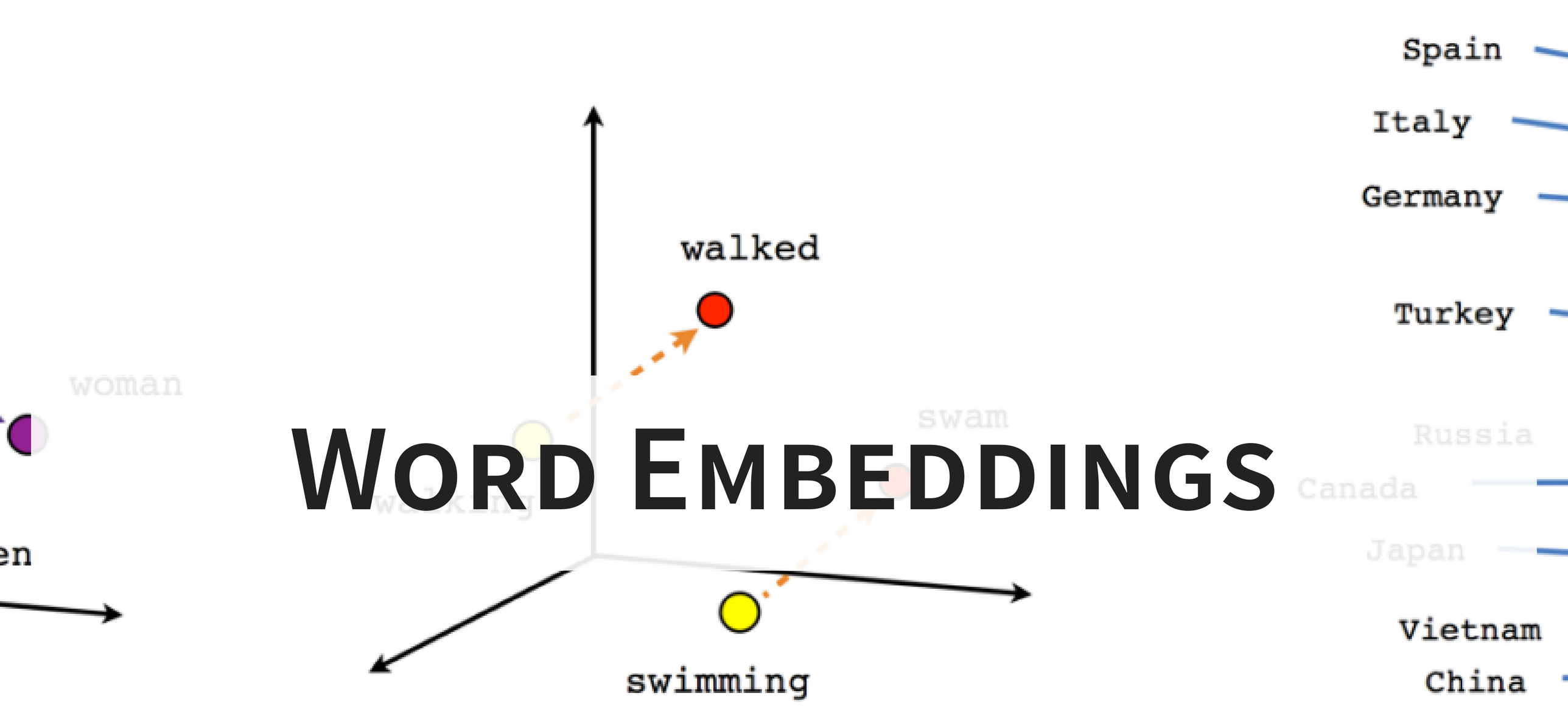
Usually these files should not be in VCS

Other systems

Other tools prefer INI or yaml systems - eg python's built in ConfigParser and Luigi

These offer great composability (eg a system file for servers/IP's, a local file for code config in VCS) and even variables in the configs

However, they often are not overrideable via the environment - an annoyance at best



Word Embeddings

Verb tense

A staple in NLP/NLU

Old and Busted

Keyword approaches like TfIdf

$$w_{i,j} = t_{i,j} \log\left(\frac{N}{d_i}\right)$$

New Hotness

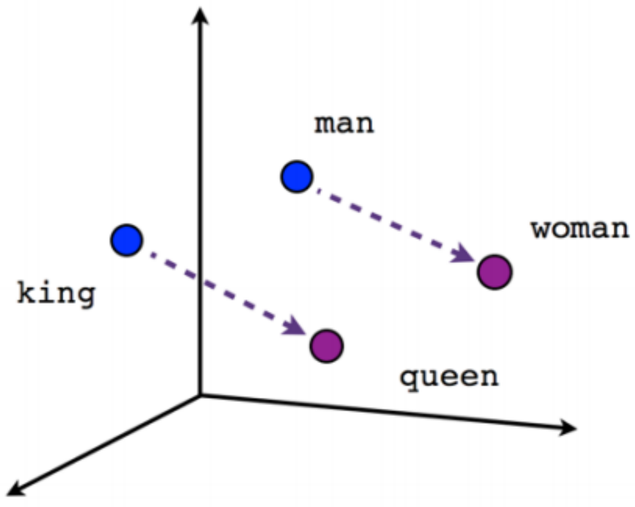
Deep representations

50~300 *dim*

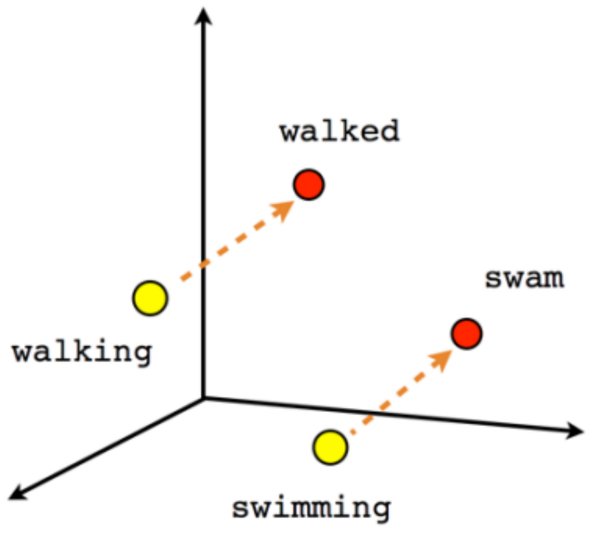
python	0.52	0.21	0.37	…
ruby	0.48	0.21	0.33	…
word	0.05	0.23	0.06	…

Magic Dimensions

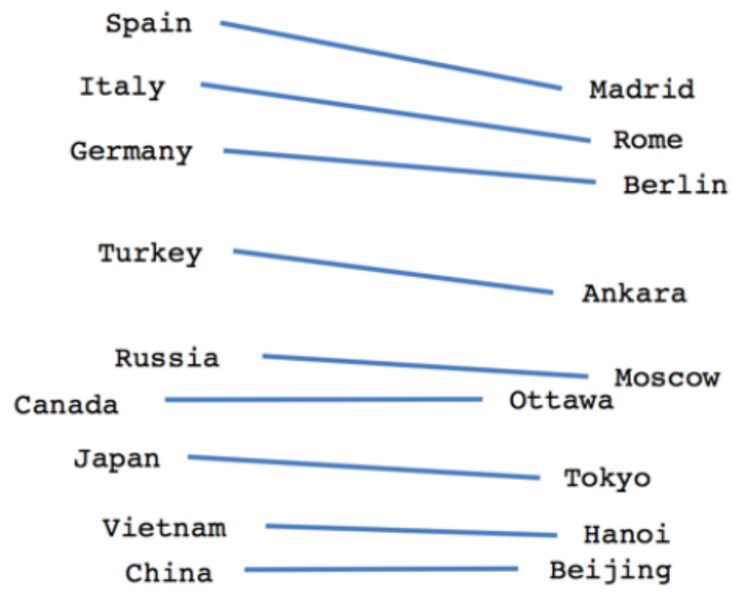
Through deep learning and dimensionality reduction, we can visualize a learned mapping of words in a vector space



Male-Female



Verb tense



Country-Capital

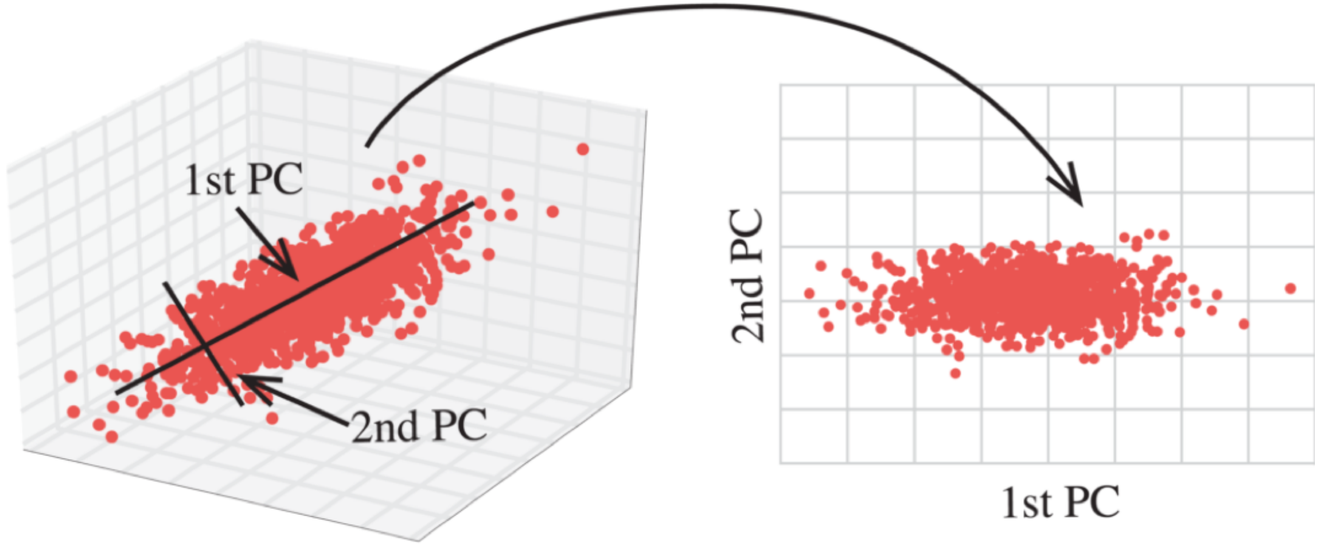
Magic Dimensions

Deep learning and NLU is beyond the scope of this course. However, we will touch aspects since they are highly relevant and interesting.

Dimensionality Reduction

We can’t imagine more than 3 dimensions, so we must project!

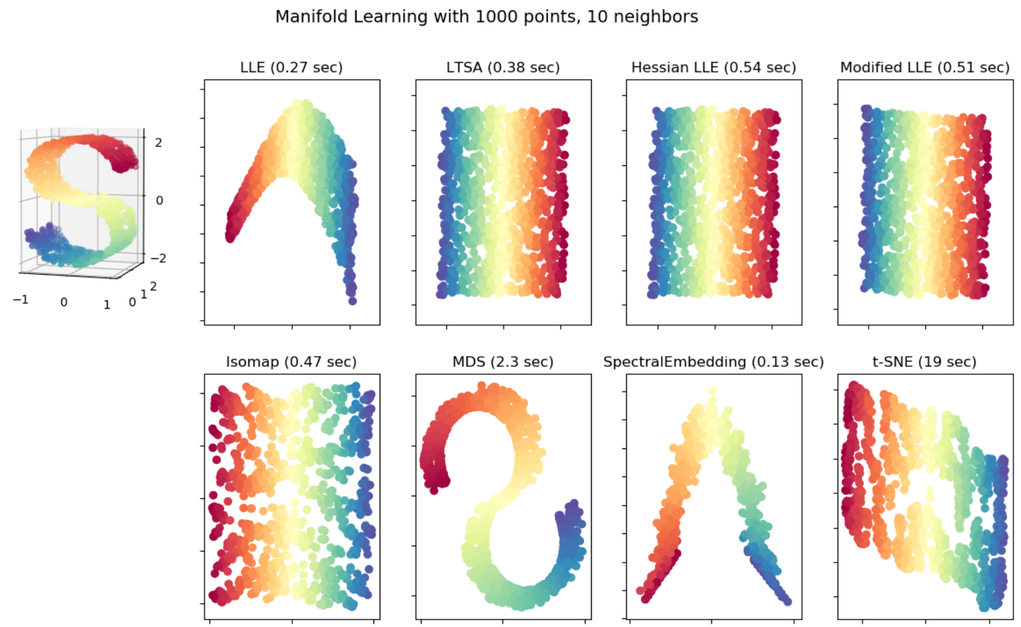
By definition, we lose information… but preserve relative distances



Dimensionality Reduction

Generally, must sacrifice:

- Accuracy
  - point -> volume
- Meaning
  - Unitless dimensions
- Aesthetics
- Working in a static image



Readings

- 12 Factor Apps
- Manifold Learning
- Visualizing MNIST