Workflows

Keeping Things Together

Dr. Scott Gorlin

Harvard University

Spring 2021

Speaker notes

No notes on this slide.

Agenda

- What is *Python*?
- (Ana) Conda
- Config
- Decorators
- Bootstraping
- Readings

Speaker notes

No notes on this slide.

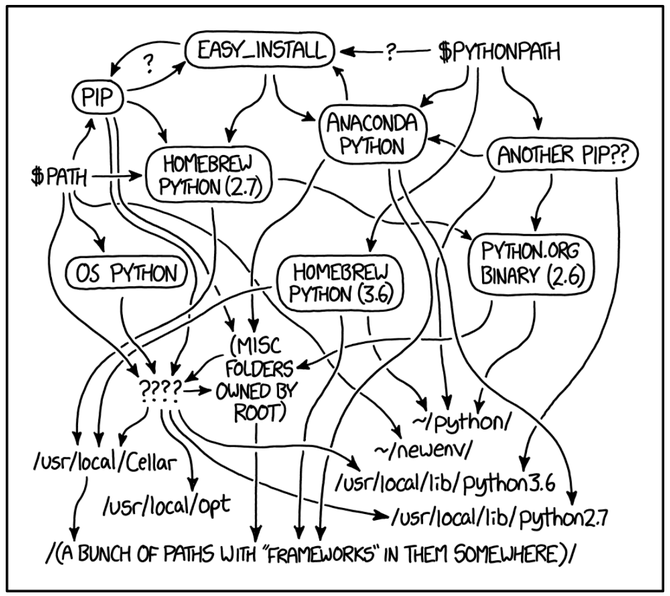
```
primes(int nb_primes):
cdef int n, i, len_p
cdef int p[1000]
if nb_primes > 1000:
    nb_primes = 1000

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 %
            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*?

Speaker notes

No notes on this slide.

You likely have 3



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

XKCD 1987

Speaker notes

No notes on this slide.

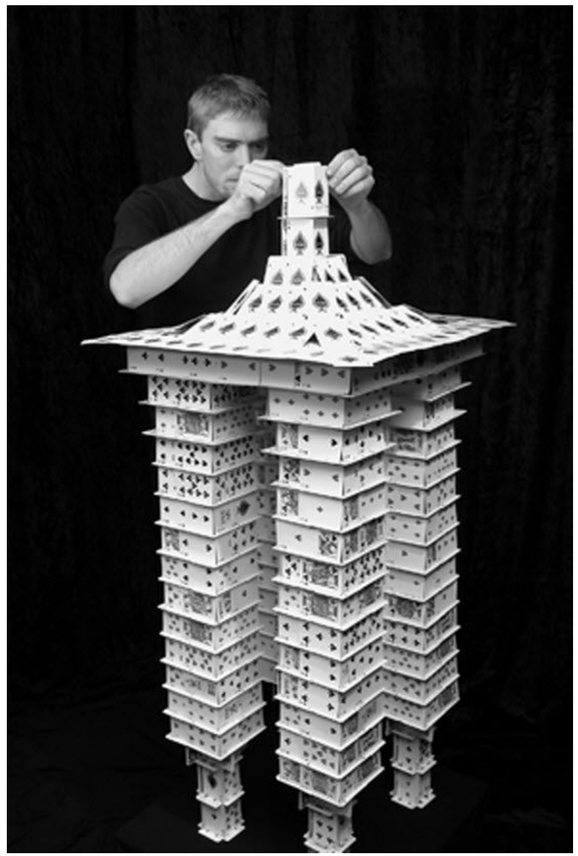
... EVEN IGNORING 2.X VS 3.X

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

Speaker notes

No notes on this slide.

What's wrong with your root python?



Your root python is a finely balanced deck of cards, in service of many

We have no way of ensuring changes for one project don't break another, like incompatible numpy requirements, or newly introduced bugs.

We have no way of replicating your environment for someone else's use.

Speaker notes

No notes on this slide.

New versions are double-edged swords

Numpy released a new version! Hooray!

New features! Awesome!

Hmm, now this test fails… is this important?

They fixed a bug… but now my workaround needs tweaking!

Uh oh, a new bug!

This other library specified an older version… now I can't rebuild!

Speaker notes

While we want to accept progress from other projects, we need to ensure we have a safety net to prevent regression in our own.

New code doesn't always 'work' seamlessly. Upgrades need to be deliberate, or at least have a way of safely rolling back. Your code environment must be tracked and managed with the same rigor as your actual code (more, actually, since you likely don't understand everything in the environment and can't fix most of it!).

Whether deploying your code to a server farm or simply checking back on a project from several months ago, we need the ability to capture and 'deploy' the same environment for the code you were working on. Python does not do this well by default, so we need some tools and workflows.

The Vision

We want a `python` that:

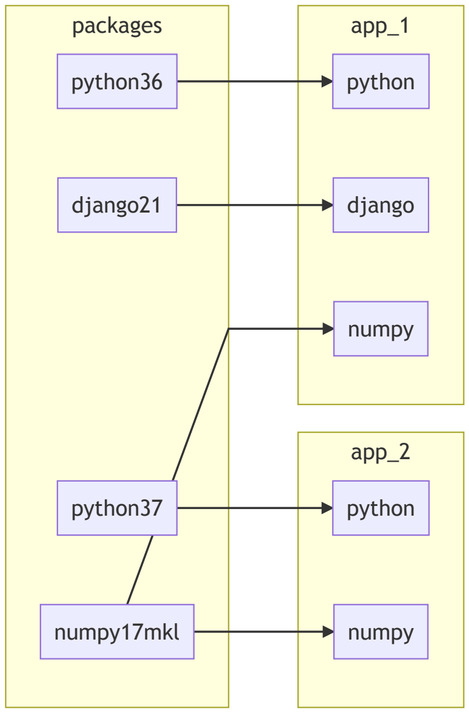
- Is specified in code and VCS
- Has everything (and only what) you need
- Won't randomly break, easy to 'roll back'
- Is the same across machines
- Your buddy can use
- Can be deployed on a production server
- Does not require you to manually install

Speaker notes

No notes on this slide.

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

Speaker notes

No notes on this slide.

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*

Speaker notes

No notes on this slide.

Environments, Apps, Libraries

Root Environment

An ‘R’-like ***global*** set of tools, which you hope will work together seamlessly (they won’t!)

App Environment

A ***specific, minimal, exhaustive*** set of dependencies to make one project work repeatably

Library "Environment"

Designed to deploy into *another* environment; specifies ***abstract*** set of deps, eg `numpy>=1.0`

Library Dev App

A special app used just to develop your library

Speaker notes

No notes on this slide.

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

Library App

… per lib

Speaker notes

No notes on this slide.

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 site packages

The Scientist:

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

- Need to juggle conda vs pip

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

Speaker notes

Note all three of these mechanisms define the build within a specification file, whether that is requirements.txt, environment.yml, or a Pipfile. It is not appropriate to manually `pip install ...` because this is not repeatable.

Even inside a coded build system that includes a `pip install ...` call, consider replacing it w/ a declarative build specification.

What are your reqs?

Abstract Library

Eg, what's in your `setup.py`:

```
setup(
    install_requires=[
        'requests[security]',
        'flask',
        'gunicorn==19.4.*',
    ]
)
```

```
(dev) python setup.py develop
(dev) pip freeze > requirements.txt
```

Frozen App

Eg, `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
```

Speaker notes

Requirements come in two forms:

Abstract
: The packages you need, specifying versions only if you need a feature from a recent release, or if their are known incompatibilities

Frozen
: An exhaustive set of every installed package, with its version, from a known working build. This may vary slightly by system, but ideally is 100% repeatable.

One (manual) way to keep both up to date is to specify abstract dependencies in a project or library's setup.py, and periodically run `setup.py develop` and/or `pip install -U ...` (but never adding new packages directly). You can then freeze the current environment with `pip freeze > requirements.txt`.

Alternatively, specify everying via two files: `pip install -r abstract_reqs.txt` and `pip freeze > frozen_reqs.txt`.

Pipenv

```
pipenv install numpy
```

Abstract Pipfile:

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

[packages]
numpy = "*"
# Or, from ./setup.py!
my_package = {editable = true, path = "."}

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

Frozen Pipfile.lock:

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

Speaker notes

Pipenv basically handles the former by dividing specs into an abstract Pipfile and freezing the environment into Pipfile.lock. Adding or upgrading packages automatically maintains both files.

You can install requirements from your package if you are developing one. The above example says ‘install `my_package` in editable/development form from the current directory’ and will pull in any requirements automatically.

Pipenv

Why?

```
pip install numpy pandas pyarrow
...
```

… is a sequential process

Initial versions will be upgraded for newer reqs, even if you don't want them!

```
pipenv install pandas pyarrow
```

… uses a graph of joint dependencies

```
(2019sp-pset-1-gorlins) bash-3.2$ pipenv graph
awscli==1.16.101
  ...
pandas==0.24.1
  - numpy [required: >=1.12.0, installed: 1.16.1]
  ...
pyarrow==0.12.0
  - numpy [required: >=1.14, installed: 1.16.1]
pytest-cov==2.6.1
  ...
xlrd==1.2.0
```

Speaker notes

No notes on this slide.

Pipenv

Why?

```
pipenv update
```

… one-stop upgrade of lock w/o breaking abstract deps

Speaker notes

No notes on this slide.

Environments

Pipenv gives you two ‘environments’:

Packages

“What you need to run”

- numpy
- tensorflow

Dev Packages

“What you need to run tests”

- pytest
- mock

Missing: IDE Packages?

… what about things like ipython and black?

Speaker notes

Pipenv explicitly separates ‘real’ and ‘development’ dependencies. Real deps are what your code actually needs (eg, things you import, like pandas or numpy).

Dev packages are a bit more undefined. They should include things you need to package up an install, or run unittests, but that are not required by the application itself. You should be able to pass all CI/CD tests doing a `--dev` install.

Pipenv doesn’t make package environments configurable, however. There is not an easy way to ensure things like black, ipython, sphinx, etc are available for code development, EDA, documentation building, etc in a repeatable way. Some of these tools may be better in your root environment (eg, black) but projects could legitimately use a way to specify multiple environment extensions.

Pycharm nicely integrates ipython with a pipenv environment automatically, but I have not discovered a simple way of adding user packages to a pipenv environment!

You can try installing pipenv with ‘site packages’ available from your base system, but this is kind of clunky and as of early 2020 has a known bug (with an unreleased patch): https://github.com/pypa/pipenv/issues/3718

Speaker notes

No notes on this slide.

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

Speaker notes

No notes on this slide.

Conda

The Good

```yaml
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 are arch. specific (appnope)

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

The Ugly

```yaml
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

Speaker notes

Conda envs are specified with an `environment.yml`, which is very powerful.

It is like a tricked-out requirements.txt. In addition to specifying version numbers, you can even specify a specific build of a package, which may be important if it it compiled differently (eg numpy using OpenBLAS vs Intel MKL which can make a difference for high performance linear algebra).

The flip side of this specificity - builds tend to be architecture specific (e.g., OSX vs Win vs linux), which makes it harder to share specifications across machines of different architecture.

It can also specify installs via conda or pip, but the two package repos offer competing 'sources of truth' for python packages!

Better/worse, conda has 'channels', which offer more choices (and more competing distributions for any package version.)

Deps in conda only install their reqs via conda, deps in pip only install their reqs via pip :(

Some developers only maintain their packages in one package manager!

Conda

The Ugly

`environment.yml`:

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

`conda env create -f`:

1. Conda installs `tornado`
2. Pip installs `flower`
    1. `flower` requires an older `tornado`
    2. `flower` install ***downgrades*** `tornado`
3. Conda requirements are not satisfied, but no error raised

Speaker notes

This is a real example I dealt with using `flower`, a visualizer for Celery, a popular messaging system and distributed task queue.

The biggest issues occur if a package dependency is specified differently from a conda install vs a pip install, eg the `flower` example above. Because pip occurs ***after*** the conda install, the conda requirement is overridden!

This not only left my system with the wrong version of tornado, it was also a verion installed from pip, not conda - very different from what was specified in the yaml.

The Unsolved

Laptop

```
name: myenv
dependencies:
  - tensorflow
```

Server

```
name: myenv
dependencies:
  - tensorflow-gpu
```

How do you create a deterministic build if the highest level components differ across hardware?

Maybe specifiers?

Speaker notes

Having ‘a deterministic build’ is impossible if you have multiple systems with legitimately different requirements!

A real example I deal with is CPU vs GPU machines for machine/deep learning - I need different environments for training and inference, depending on the problem and the model. Ideally, I could ‘conda install tensorflow’ and the packages would run on GPU or CPU as needed; however, conda has different package names for different backend dependencies!

This is a problem for pip/pipenv as well. There is no single principled way to adjust a build for multiple environments; having 2 separate environments works, but then there is no way to force them to use the same common packages, eg numpy. There are many workarounds, but no ‘golden’ solution.

The Promising

Anaconda Project looks a lot like Pipenv for conda!

It even supports multiple custom environments, eg: default, dev, eda, gpu…

… except ~~the last release was in 2017~~ adoption is sparse

Speaker notes

No notes on this slide.

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

Compiled binary wheels on PyPI for things like numpy make conda much less critical (though it still is better)

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

Speaker notes

No notes on this slide.

What else?

See packaging tutorial

Poetry

Better for pip-installables (eg libraries)

Blocker Does not support setuptools-scm

Pip-tools

Internal tools used by pipenv, build your own workflow

µPipenv

Cross compiles poetry/pipenv?

Speaker notes

No notes on this slide.

My Recommendations

Use Pipenv

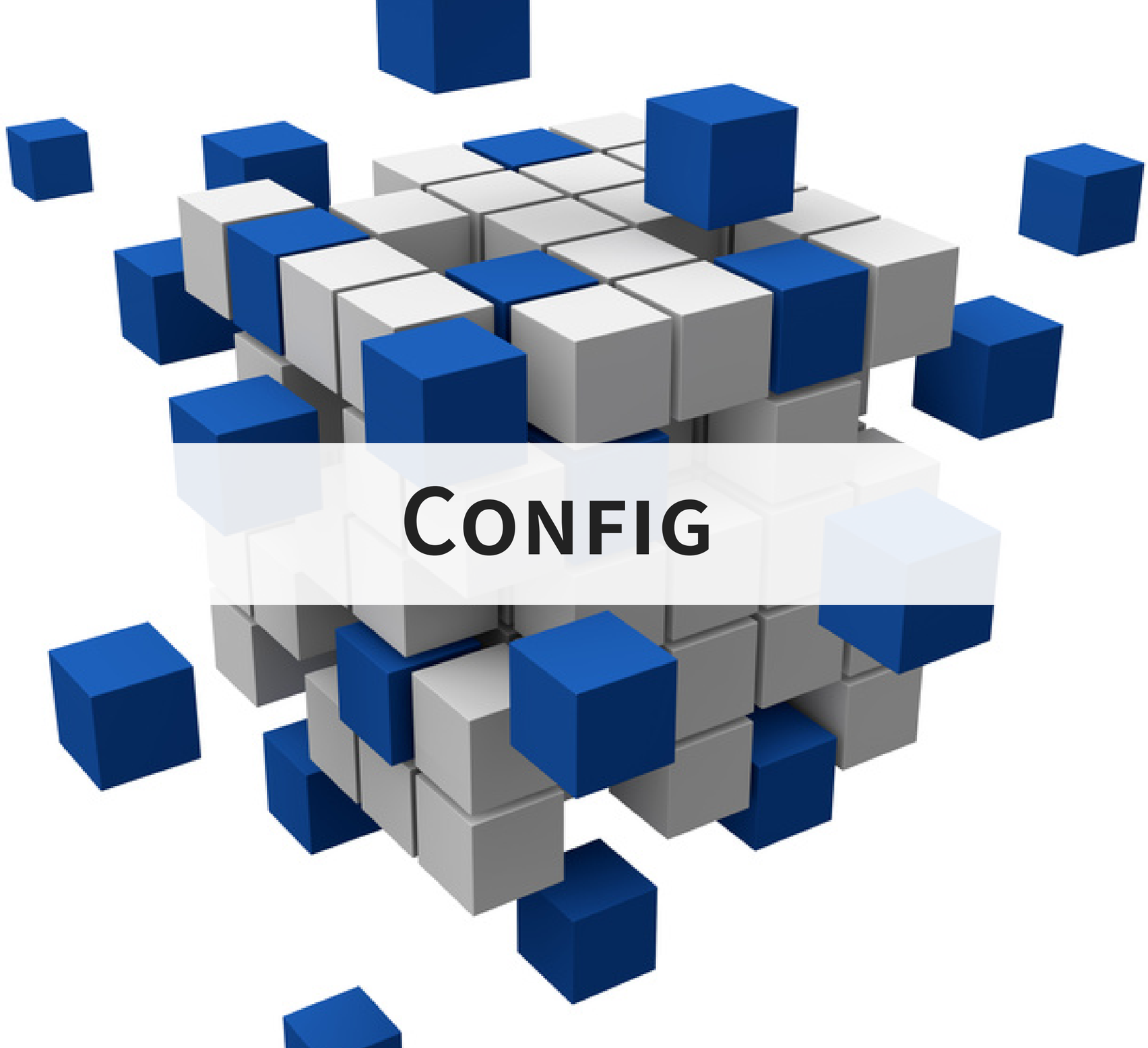
- For pure-python projects
- … or basic pydata stack
- … or with Docker if you don’t mind slower rebuilds, or have a robust container registry

Use Conda

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

Speaker notes

No notes on this slide.



Config

Speaker notes

No notes on this slide.

Your code is like my 5 y.o. daughter

… 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*

Speaker notes

No notes on this slide.

Why Config?

We achieve 2 things with app configuration:

Security

App secrets are not leaked via code

Flexibility

New & different deploys should be possible with little to no code modification

Speaker notes

When designing an app, think about any particular design choices you make - where files are stored, how you fetch or connect to a system, etc. Ask yourself which of these ***are likely to change*** - whether by running on a new system, or in a test environment, etc.

Anything that reflects a configuration must be removed from the code, especially if it is sensitive (authentication details) but just for good practice as well.

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.

```
load_model('/Users/scott/Documents/my_model.pkl')
```

```
load_model(
    os.path.expanduser(
        "~/Documents/my_model.pkl"))
```

```
DATA_DIR = os.environ.get(
    'DATA_DIR', os.path.join('.', 'data'))

load_model(
    os.path.join(DATA_DIR, 'my_model.pkl'))
```

Speaker notes

Even generalizing the username in the right example assumes that the user has a `Documents` folder and wants the data there. A linux server likely would not have this folder.

Allowing for config through the environment is an extremely easy way to provide flexibility to multiple deploy environments. You can still set reasonable defaults as above.

Absolute Nothing

```
canvas = Canvas(
    "https://canvas.harvard.edu", "<token>"
)
course = canvas.get_course(12345)
```

```
course =
canvas.get_course(os.environ["COURSE_ID"])
```

```
HOST = os.environ.get(
    "CANVAS_URL",
    "https://harvard.test.instructure.com/")
...
course = find_course(canvas.get_courses(),
**kwargs)
pset = find_pset(course, 'Pset 1')
```

Speaker notes

The more ‘portable’ your code is, the better. Eg, private ID’s, like the course id in canvas, may be stable - but they are not portable. Imagine using the Testing canvas installation, or taking the course again in a new semester. Ideally, you don’t need to recode anything - and even better, if you can avoid reconfiguring by dynamically loading any private ID’s using more ‘natural’ and portable identifiers, the better.

... 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.

Speaker notes

No notes on this slide.

Config Basics

You can simply read any string value:

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

Or you may need to parse to 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'
    )
)
```

Speaker notes

No notes on this slide.

Config Basics

Look for frameworks to do the heavy lifting…

```
from environs import Env
env = Env()

COURSE_ID = env.int("COURSE_ID")
```

```
from envparse import Env
env = Env(
    BOOLEAN_VAR=bool,
    LIST_VAR=dict(cast=list, subcast=int))
```

Speaker notes

No notes on this slide.

Config Basics

Schema's are nice…

Revisit this once we cover descriptors!

```
class EnvVar:
    def __init__(self, field_type="str", **kwargs):
        ...

    def __set_name__(self, owner, name):
        ...

    def __get__(self, env, objtype=None):
        return getattr(env, self.field_type)(
            self.get_name(env), **self.kwargs)
```

```
class CANVAS(Env):
    TOKEN = EnvVar()
    COURSE_ID = EnvVar('int')
```

Speaker notes

No notes on this slide.

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.

Speaker notes

No notes on this slide.

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.

We will use and recommend dotenvs for your psets in this course

Speaker notes

No notes on this slide.

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.

Speaker notes

No notes on this slide.

Extra

Look to frameworks when doing substantial configs, eg:

environs

```python
from environs import Env

env = Env()

# export MYAPP_HOST=lolcathost
with env.prefixed("MYAPP_"):
    host = env("HOST", "localhost")

# simple validator
env.int("TTL", validate=lambda n: n > 0)
```

A good config system provides:

- Config typing (int, bool, etc)
- Organization (all env vars defined in one spot)
- Eager or lazy validation
- Env overrides

Speaker notes

No notes on this slide.

Decorators

Speaker notes

No notes on this slide.

Decorating Classes

```
@decorator
class A:
    ...
```

```
class A:
    ...
A = decorator(A)
```

The syntax is the same, but your decorator may need to be designed for classes

Speaker notes

The biggest obstacle to writing decorators for functions vs classes is the `self` attribute passed to all instance methods - depending on the decorator, it may or may not impact resulting code. See later lectures on cacheing for an example

Wrapping Classes

```python
def make_child(cls):
    class ChildClass(cls):
        if some_logic():
            def x(self):
                return 1
        else:
            def x(self):
                return 2
    return ChildClass

@make_child
class Parent:
    ...
```

Wrapping classes allows for dynamic coding with extreme sophistication.

This is a bit too complex for now. We may cover later in semester.

Speaker notes

Note here we:

1. dynamically generate a new child class, like a ‘wrapped’ function
2. define the class differently depending on the logic/state within the decorator.

This is one method of metaprogramming.

Note how this differs subtly (but substantially) from:

```python
class ChildClass(ParentClass):
    def x(self):
        if some_logic():
            return 1
        return 2
```

The Wrapper Pattern

Preprocessing

```
def as_bytes(func):
    @wraps(func):
    def wrapped(*a):
        return func(*[
            x.encode()
            for x in a
            if isinstance(x, str)
            else x
        ])
    return wrapped
```

Usage

```
@as_bytes
def something(arg: bytes):
    # Assume arg is now bytes
    ...
```

The function can now accept `bytes` or `str` safely!

Speaker notes

When we preprocess arguments in a way that changes their type, we are left with a minor dilemma regarding the documentation of the function!

Here, the documentation states the function can handle string or bytes:

```
def something(arg: Union[str, bytes]):
    if isinstance(arg, str):
        arg = arg.encode()
    ...
```

vs here, we indicate the same - but only because of the decorator! The function itself can only handle bytes:

```
@as_bytes
def something(arg: Union[str, bytes]):
    # Assume arg is now bytes
    ...
```

vs here, where the docs are accurate wrt the original function, but not the wrapped function everyone actually uses.

```
@as_bytes
def something(arg: bytes):
    # Assume arg is now bytes
    ...
```

Technically, it would be possible for the decorator to intercept the type hints and modify them on the wrapped function, to get the best of both worlds. However, if you are using vanilla docstrings, this would be much harder.

What's with `@wraps(func)`?

This is a ***parameterized decorator!***

Speaker notes

No notes on this slide.

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)
```

Speaker notes

No notes on this slide.

Decorators – With Args

… or more clearly…

Vanilla

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

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

Parameterized

```
@decorator_generator(param='val')
def func(x):
    ...

# Equivalent to
func = decorator_generator(param='val')(func)
```

Speaker notes

Here, `decorator_generator` is generating a function/object internally that will be the 'true' decorator.

We need this because, when the decorator is called, it only gets the single function as an argument! Things would be a lot simpler if we could explictly form the call to the decorator while still using the `@` syntax, but python does not provide this capability.

Decorators – With Args

Why??

```
@decorator_generator(param='val')
def func(x):
    ...

# Equivalent to
func = decorator_generator(param='val')(func)
```

Note this is more explicit:

```
func = decorator(func, param='val')
```

… but cannot be achieved with the @ syntax!

Speaker notes

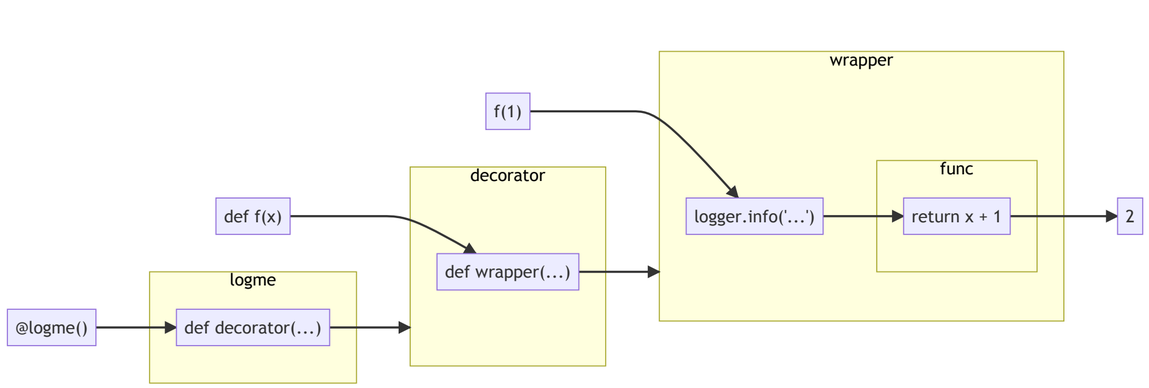
The @ calling syntax only receives the function/class to decorate; the syntax does not allow us passing in args or kwargs.

The decorator generator does not receive the function to decorate. Therefore, it must 'encapsulate' the params into a special decorator it generates (for the sole purpose of encapsulating said parameters).

This is arguably syntactic saccharine, since we now need more complicated code when using the @ syntax!

Decorators – With Args

Parameterized decorators must encapsulate twice!




```python
def logme(logger=root):
    # Must generate a decorator
    def decorator(func):
        @wraps(func)
        def wrapper(*a, **kw):
            logger.info(...)
            return func(*a, **kw)
        return wrapper
    return decorator
```

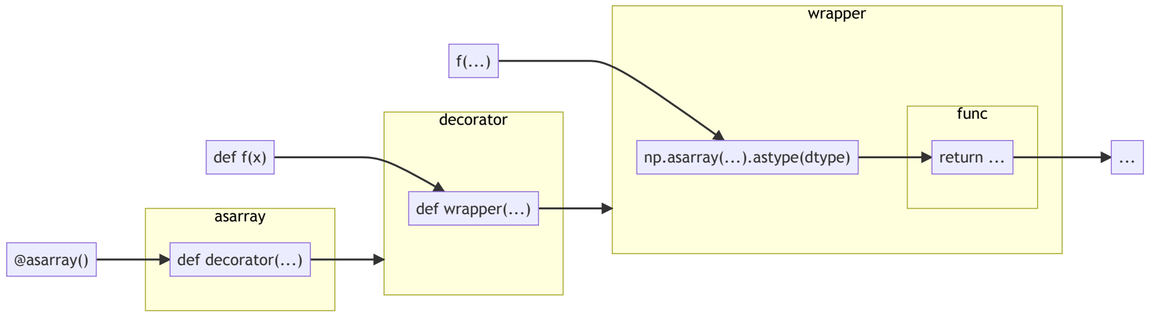
```python
@logme()
def f(x):
    ...
```

Speaker notes

No notes on this slide.

Decorators – With Args

Parameterized decorators must encapsulate twice!




```
def asarray(dtype=float):
    # Must generate a decorator
    def decorator(func):
        @wraps(func)
        def wrapper(*a, **kw):
            # use np.astype(dtype)
            ...
        return wrapper
    return decorator
```

```
@asarray() # '@asarray' fails!
def f(x):
    ...

@asarray(dtype=int)
def g(x):
    ...
```

Speaker notes

No notes on this slide.

A LOOK AT `wraps()`

```
from functools import partial, wraps

@wraps(func)
def wrapper(...):
    ...
```

```
def wraps(wrapped):
    # Use a partial call as the decorator!
    return partial(update_wrapper, original=wrapped)

def update_wrapper(wrapper, original):
    wrapper.__doc__ = original.__doc__
    ...
    return wrapper
```

We create a ***closure*** with the function, pushing it into a ***future*** call (the decorator) using `partial`.

```
def partial(func, *args, **kwargs):
    def closure(*exa, **exk):
        newk = kwargs.copy()
        newk.update(exk)
        return func(*args, *exa, **newk)

    return closure
```

Speaker notes

These are not the real implementations - they are simplified for discussion.

Using `partial` is a great way to 'partially call' a function - it is not called now, but rather, we create a new function that, when called in the future, contains some of the arguments already filled in. For keyword arguments, you can think of this as changing the default values. Really, we are partially setting all the arguments now, and waiting for the function to finally be called with the rest of the args.

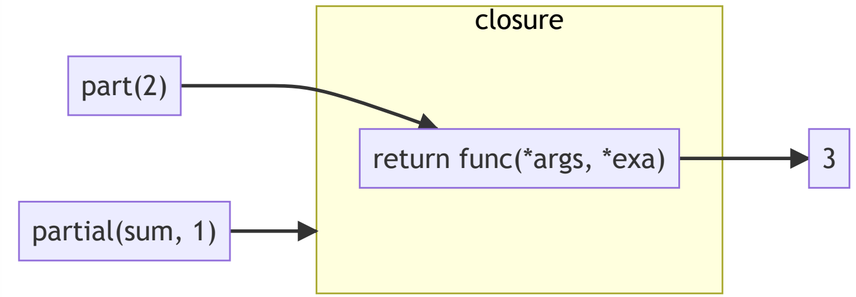
In this way, `wraps` can grab the function to be wrapped, but immediately return a decorating function (since it cannot generate the wrapper function directly).

A LOOK AT `wraps()`

We create a ***closure*** with the function, pushing it into a ***future*** call (the decorator) using `partial`.

```
def partial(func, *args, **kwargs):
    def closure(*exa, **exk):
        newk = kwargs.copy()
        newk.update(exk)
        return func(*args, *exa, **newk)

    return closure
```

```
part = partial(sum, 1)
part(2) # 3
```



Speaker notes

No notes on this slide.

Decorators – Uniting Parameters

We can unite parameterized decorators by juggling keywords!

```python
# Raw decorator, func=f
@asarray
def f(x):
    ...

# Decorator generator, func=None
@asarray(dtype=int)
def g(x):
    ...
```

```python
def asarray(func=None, dtype=float):
    if func is None:
        return partial(asarray, dtype=dtype)

    @wraps(func)
    def wrapped(*args, **kwargs):
        ...
    return wrapped
```

Speaker notes

This implementation detects whether the function is used directly as a decorator (`func` will be a callable) or as a decorator generator (`func` is `None`), which generates a decorator via a `partial` call to itself!

The pattern is generalizeable, and can be extracted as a `def decorator_with_kwargs(func)` helper function for decorators!

Decorators – Uniting Parameters

Encapsulate best practices in code!!

```
def decorator_with_kwargs(decorator):
    @wraps(decorator)
    def wrapper(func=None, **k):
        if func is None:
            # Encapsulate kwargs, generate
            # a decorator!
            return partial(decorator, **k)
        # Used directly
        return decorator(func=func, **k)
    return wrapper
```

```
@decorator_with_kwargs
def asarray(func=None, dtype=float):
    @wraps(func)
    def wrapper(*a, **k):
        ...
    return wrapped
```

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

@asarray()
def g(x):
    ...

@asarray(dtype=int):
def h(x):
    ...
```

Speaker notes

As noted, decorators are very difficult syntax to get straight. This is true for you as the author, as well as any users of your code!

This single complex pattern can be used to greatly alleviate the complexities of writing - and using - parameterized decorators. Now, you can focus on a single wrapping, and your users don't have to worry as much about how to use `()` when invoking your decorators!

Note the use of `partial`, which is like a 'partially called function' - we are using it to encapsulate the kwargs already passed in. It is kind of the equivalent to:

```
def partial(func, *oa, **ok):
    def wrapper(*a, *k):
        return func(*oa, *a, **ok, **k)
    return wrapper
```

… but with more sophistication.

An Alternate - Class-Based Decorators

Class based

```
class AsArray:
    def __init__(self, dtype=float, ...):
        self.dtype = dtype
        self.targets = ...
```

```
    def __call__(self, func):
        @wraps(func):
        def wrapper(*a):
            ...
        return wrapper
```

Usage

```
@AsArray()
def process(arg: np.NDArray):
    ...
```

or

```
asarray = AsArray(targets=(pd.DataFrame,))

@asarray
def process(..):
    ...
```

Class based decorators may be cleaner than parameterized ones

Speaker notes

This is a bit cleaner for parameterized decorators because we do not have to stack a bunch of inner functions. Classes are the 'correct' way to easily design 'functions which generate functions' - basically, you just capture some parameters as instance attributes (the role of `__init__`) and apply the appropriate logic when the instance is called (`__call__`).

We create an instance of `AsArray`, which is the decorator. Recall that the `@` syntax hides the invoking `()` on the function, so `AsArray(...)` is an instance, and `@some_instance` invokes the `__call__` method of that instance.

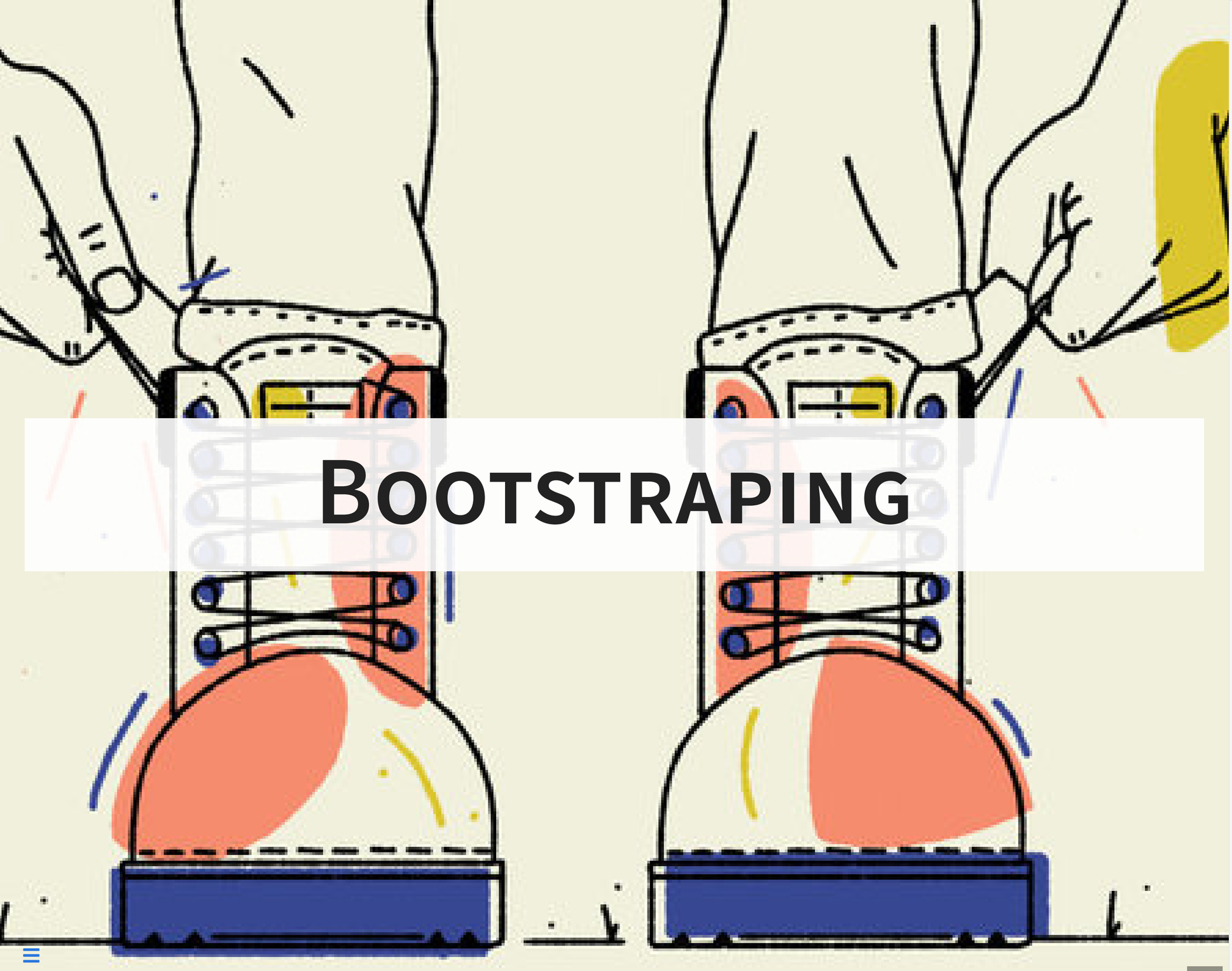
The `__call__` method then creates the wrapper function and returns it.

The only reason not to follow this pattern is if you want to allow using the decorator without calling it directly (`@logme` vs `@logme()`) for simpler user syntax.

Note that, for whatever reason, many package authors use this methodology but don't follow standard naming conventions - we often see classes like:

```
class some_decorator_class:
    ...
@some_decorator_class(...)
def f(...):
    ...
```

There is no reason not to use standard python naming conventions like `SomeDecoratorClass`.



Bootstraping

Speaker notes

No notes on this slide.

Jinja2

Variable rendering…

```
>>> 'Hello {name}!'.format(name='world')
'Hello world!'
```

```
>>> from jinja2 import Template
>>> template = Template('Hello {{ name }}!')
>>> template.render(name='John Doe')
'Hello John Doe!'
```

Speaker notes

Jinja2 is a popular templating language in Python. It basically lets you run generic python code to configure text output.

Cookiecutter uses Jinja for rendering

Jinja2

… plus full python logic!

```
<ul>
{% for user in users %}
  <li><a href="{{ user.url }}">{{ user.username }}</a></li>
{% endfor %}
</ul>
```

```
Template(...).render(
    users = [{...}]
)
```

Templates help to separate the ‘view’ (and view logic) from ‘data’!

Speaker notes

Thinking about your output as a template - or view of the data - is very powerful, if you can then simply feed data to the template for rendering. This paradigm helps isolate responsibilities for data collection, specifying the actual view, and rendering the view. We’ll explore this - MVT and MVC paradigms - more formally later.

Cookiecutter

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)

Speaker notes

No notes on this slide.

Cookiecutter

Better Project Templates

```
cookiecutter-something/
├── {{ cookiecutter.repo }}/  <-- Project template
│   ├── {{ cookiecutter.project_name }}
│   │   └── __init__.py
│   └── setup.py
├── blah.txt                  <------ Non-templated files/dirs
│                                     go outside
│
└── cookiecutter.json         <------ Prompts & default values
```

Speaker notes

No notes on this slide.

Cookiecutter

Encapsulating Best Practices

src/{{ package }}/cli.py:

```
{%- if cookiecutter.command_line_interface == 'click' %}
import click

@click.command()
@click.argument('names', nargs=-1)
def main(names):
    ...
```

```
{%- elif cookiecutter.command_line_interface == 'argparse' %}
import argparse

parser = argparse.ArgumentParser(description='Command description.')
parser.add_argument(...)

def main(args=None):
    args = parser.parse_args(args=args)
    ...
{%- endif %}
```

Speaker notes

No notes on this slide.

Cookiecutter

Code of Conduct

A Pantry Full of Cookiecutters

- Python
  - Cookiecutter (meta)
  - Ansible
  - Git
  - C
  - C++
  - C#
  - Common Lisp
  - Elm
  - Golang

A Pantry Full of Cookiecutters

Here is a list of **cookiecutters** (aka Cookiecutter project templates) for you to use or fork.

Make your own, then submit a pull request adding yours to this list!

Python

- cookiecutter-pypackage: @audreyr's ultimate Python package project template.
- cookiecutter-pipproject: Minimal package for pip-installable projects
- cookiecutter-pypackage-minimal: A minimal Python package template.
- cookiecutter-lux-python: A boilerplate Python project that aims to create Python package with a convenient Makefile-facility and additional helpers.
- cookiecutter-flask : A Flask template with Bootstrap 3, starter templates, and working user registration

Speaker notes

There are dozens of available cookiecutter templates for many different project types. When starting something new, see if one helps!

Cookiecutter

...including Cookiecutter Data Science

Cookiecutter Data Science

Why use this project structure?

Other people will thank you

You will thank you

Nothing here is binding

Getting started

Requirements

Starting a new project

Example

Cookiecutter Data Science

A logical, reasonably standardized, but flexible project structure for doing and sharing data science work.

Why use this project structure?

> We're not talking about bikeshedding the indentation aesthetics or pedantic formatting standards — ultimately, data science code quality is about correctness and reproducibility.

Speaker notes

This template is well worth exploring. Their post has some great ideas and is clearly written.

However, I find it not fully functional for my brand of DS work, or this class.

Some real issues:

- No such clean division between raw, external, processed, final data etc
- Notebooks really should not be a main part of your repo
- Other, similar things should not be in VCS: references, models, etc
- Don't just 'sync data to s3' (multiple buckets, too big to sync, etc)
- `make` and `Makefile` work well for compilation and simple local projects, but not for distributed/large scale/complex work
- Naming the package `src`!

Readings

Speaker notes

No notes on this slide.

Now

- Python Packaging
- Functional Programming HOWTO
- So you want to be a functional programmer
- Cookiecutter Data Science

Upcoming

- Luigi

Optional Word Embeddings

- Manifold Learning
- Visualizing MNIST

Speaker notes

No notes on this slide.