Writing Python for Reproducable Research

Boston Python Presentation Night

Graham Voysey

23 October 2018

Outline

Who Am I?

Ethos

Perverse Incentives Preserve Old Challenges

Research Code Makes Reproducability Harder

Code Provenance

Two Kinds of Data Provenance

My Motivating Question

Oh, One More Thing...

Who Am I?

Slides

These slides are now online at github.com/gvoysey/talks

Bio

- Research Engineer at Boston University's Hearing Research Center
  - Computational models for hearing in humans and whales
- Research Scientist at Neurala
  - Deep Learning for Speech Recognition, Computer Vision
- Coding in industry and academia since 2004
  - Python since 2013

Contact

- freenode: `grym`
- email: gvoysey@bu.edu
- github: github.com/gvoysey

Ethos

Enlightened Self-Interest

The “reproducability crisis” in a nutshell:

As computational scientists and researchers:

- We write code.
- We get numbers and plots out.
- We change our code!
- We change our data!
- Someone needs the old data regenerated.

...now what?!

Research results are heavy and important

Output is costly to compute

- long training jobs (days to weeks).
- long simulations (weeks to months).

Output is valuable by itself

- Publication figures.
- Detailed statistical results.

Output is divorced from the code that made it

- Needs provenance.
- Needs to be related back to what made it.

Perverse Incentives Preserve Old Challenges

We need to be able to reproduce results reliably.

This is essential to the scientific endeavor.

It's important in industry, too.

We routinely fail to reproduce results reliably.

Nobody trains scientists to code, or focus on development reproducability.

Our systems do not reward reproducability.

Cognitive biases reinforce cargo cults.

Failure to reproduce has consequences.

Papers get retracted.

Research findings fail to replicate.

Investors become... *displeased*.

Research Code Makes Reproducability Harder

Sympathy for the Devil

This is only partly culturally driven.

I think tooling is the low-hanging fruit.

Tooling won't solve everything.

General advice

This is really its own ~~rant~~ talk, so briefly, my top 3:

Write smaller and more functions

- See Brandon Rhoades' The Clean Architecture in Python
- Prefer small, easy to test pure functions.
- “Neuter I/O by promoting it to management”
- (don't inline `matplotlib` in business logic!)

Use sane variable names.

Please no:

```
from numpy import *
PL = (((beta - theta2 * theta3) / theta1) - 1) * PI2
PG = 1 / (theta3 - 1 / PL)
VL = theta1 * PL * PG
CI = spont / PI1
CL = CI * (PI1 + PL) / PL
```

Write tests.

- You have to; how else can you trust yourself?
- `pytest` and `pytest-cov` are great.
- Especially for researchers, `hypothesis` is awesome.

John Woods' Maxim

> *[...] Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.*

Normally that violent psychopath will be you, 3 months from now.

Learn your tools.

Python is idiomatic

- “When in Rome...”
- I highly recommend Jeff Knupp’s “Idiomatic Python” as a thesaurus
    - especially if you’re already very good at other languages.
    - <u>especially</u> especially if those langauges are statically typed.

Python is not “like matlab or fortran, basically”

Our tools have very particular APIs

- Pandas, Numpy, Tensorflow, Keras are all basically their own languages.
- Know their semantics!
- Let’s see how this can cause trouble...

A notable bug

```
for i in range(dim_in[0]):
        VihcNF[yc_positive] = A0 * log(1 + B * abs(yc[yc_positive]))
        VihcNF[yc_negative] = -A0 * (((abs(yc[yc_negative]) ** C) + D) / ((3 *
 ↪    abs(yc[yc_negative]) ** C) + D)) * log(1 + B * abs(yc[yc_negative]))
        VihcNF[yc_negative] = -A0 * (((abs(yc[yc_negative]) ** C) + D) / ((3 *
 ↪    abs(yc[yc_negative]) ** C) + D)) * log(
            1 + B * abs(yc[yc_negative]))

        y1 = C1LP * past_output1 + C2LP * (VihcNF + past_input1)  # intermediate
 ↪    output of the iir cascade
        v = C1LP * past_output2 + C2LP * (y1 + past_output1)
        #    # update filters' past values
        past_input1 = VihcNF
        past_output1 = y1
        #    #the output is store on the v variable
        past_output2 = v
        ihcout[i, :] = v
```

Limit jupyter notebook use

See Joel Grus' "I Don't Like Jupyter Notebooks" for The Definitive Rant.

- It's easy to make stale cells carry confusing state.
- Versioning is hard.
- Testing is more or less impossible.
- Jupyter is <u>great</u> for making summary PDFs
- Jupyter is <u>great</u> for exploring how to plot things.

Software Version Control

Used way less than you'd think.

- "Oh, it's just a little script".
- "Oh, it's a notebook, I'm just experimenting."
- "What's version control? We never learned that in grad school."
- "The lab fileshare is backed up, right?"

Software Version Control

5 months later...

- 12 zip files in your home directory.
- ...with dates in the names that aren't even right.
- ...with filenames like `CreateStim_ys_edit_2017_09_11_jr_kr.m`.
- "Oh I think this function works in this other folder, but I swear I moved it over yesterday!"
- An email from the VP of Sales arrives: "Our new customer could really use this; when can you ship this week?"

`__version__` s are for packages too:

`versioneer`: my secret weapon.

Auto-update your `__version__`.

- uses git tags to define a `__version__`.
- changes every time your files do.
- increment major/minor versions with `git tag`.

This is not overkill!

- ...for scientific code.
- ...even if nedbat disagrees with me :)
- (It might be for your web browser, though.)

Remember, our results are heavy.

Code Provenance

Packages should track provenance

Things to think about:

- What data am I using?
- What version of my code produced this output?
- How do I link output and code?
- Are there any libraries that I depend on I should keep track of?
- (`Pipenv.lock`)

Write a debug-output function.

- and put everything you can think of into it.
- and nest as much as you can.
- goal 1: copy-paste this into a JIRA ticket.
- goal 2: fingerprint every result thing (image, plot, whatever) with this.

Example debug function

```python
def get_metadata(self, args=None) -> Metadata:
    """Return configuration information about this instance and its dependencies."""
    return attr.asdict(Metadata(
        confidence_threshold=self.config.MIN_CONF,
        classifier_name=self.classifier_name,
        nms_threshold=self.config.NMS_THRESH,
        size_threshold=self.config.MIN_BOX_SIZE,
        mask_threshold=self.config.MASK_THRESH,
        bbox_size_lambda=self.config.BBOX_SIZE_LAMBDA,
        semantic_threshold=sem_thresh,
        class_list=self.classifier_data.class_names,
        backbone_weights=self.model_path.as_posix(),
        classifier_path=self.classifier_data_path.as_posix(),
        package_version=__version__,
        mxnet_version=mxnet.__version__,
        classifier_config= self.classifier.get_configuration()
    ))
```

Example Output

```
{
    "classifier_name": "OMEN",
    "confidence_threshold": 0.5,
    "nms_threshold": 0.3,
    "size_threshold": 0.02,
    "mask_threshold": 0.35,
    "bbox_size_lambda": 0,
    "semantic_threshold": -1,
    "maskrcnn_weights": "/home/gvoysey/Projects/weights/mask_rcnn_coco.h5",
    "irene_path": "/home/gvoysey/Projects/irene/resources/10_class_20_sample.npy",
    "irene_version": "0.3.0+12.g203719c",
    "time_utc": "2018-08-08T14:46:29.580002+00:00",
    "nemo_config": {
    "pynemo_version": "0.4.9",
    "ngap_version": "1.3.0+sha.e88a0d5c2.notag",
    "python_executable":
 ↪    "/home/gvoysey/.local/share/virtualenvs/pynemo-daaDd81q/bin/python3.6m",
    "python_version": "3.6.5 (default, Apr 24 2018, 12:32:07) \n[GCC 5.4.0
 ↪    20160609]",
    "os": "Linux-4.4.0-130-generic-x86_64-with-debian-stretch-sid"
    }
}
```

Two Kinds of Data Provenance

Consumed Data

“I trained my DNN on exactly this dataset”

- “...and here’s one command to regenerate it.”

`git lfs` is ...OK.

- but you do get fine grained state!

I’m open to other options.

Output data

Bake your code provenance into all your output.

Prefer automatically parsable formats

- JSON, YAML, TOML, hdf5.. whatever.
- Consider having a `--debug` command line argument that will dump this as JSON.
- Dump this as a sidecar file or to `stdout`

Output data

Embed at least `__version__` into figures, plots.

- PDF metadata (title, author, notes...)
- EXIF
- document metadata
- include a datestamp if you like.

DON'T trust filenames.

- then I just have to re-parse your non-format.

My Motivating Question

What’s our due diligence look like?

As researchers, scientists, and developers, how do we set ourselves up to make our lives easy?

What should we change about the status quo?

How much of this work can we automate?

Perverse incentives are not going to change just because I think they ought to.

Code Quality isn't either.

What clean steps *can* we take?

How should we think about this?

Developers

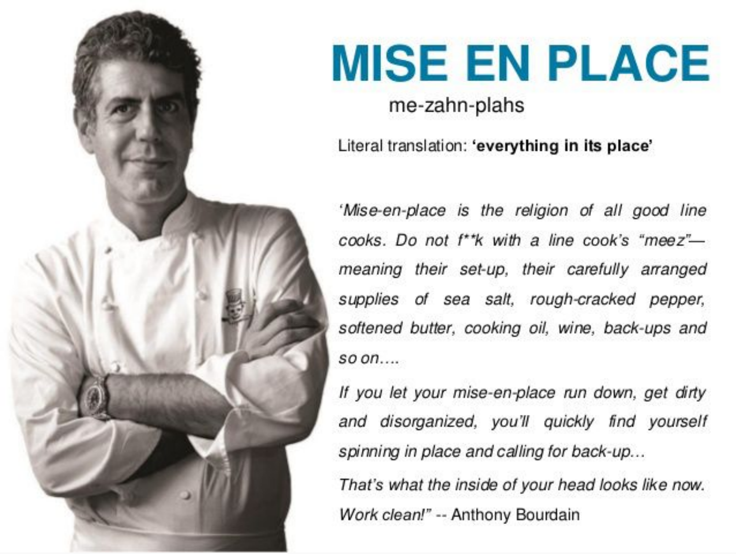
“What can I do to short-circuit this problem for my peers?”

Researchers

“What can I do to make my life easy?”

I suggest Mise en Place.

Mise en Place should be free!



MISE EN PLACE

me-zahn-plahs

Literal translation: **'everything in its place'**

*'Mise-en-place is the religion of all good line cooks. Do not f**k with a line cook's "meez"—meaning their set-up, their carefully arranged supplies of sea salt, rough-cracked pepper, softened butter, cooking oil, wine, back-ups and so on….*

If you let your mise-en-place run down, get dirty and disorganized, you'll quickly find yourself spinning in place and calling for back-up…

That's what the inside of your head looks like now. Work clean!" -- Anthony Bourdain

Oh, One More Thing...

Hey, isn't this why we invented computers?

This all sounds like a lot of work.

More to the point, it's automatic work that you're going to want to do every time you start a new project.

... isn't that why we invented computers?

Introducing `cookiecutter-python-scientific`

I've made a tool!

It will automatically generate a new project preconfigured with nearly every recommendation I've just made.

You can choose what kind of water to lead horses to.

Play with it right now.

`https://github.com/gvoysey/cookiecutter-python-scientific`

The `cookiecutter` package

Python, Jinja2-based templating system.

Consumes a configuration JSON file

- add whatever keys you like.

Emits *basically whatever*.

- you can template basically any plain-text format.

python hooks with safe cleanup.

- perform complex tasks and undo template generation if they fail.

Interactive project generator.

- user prompt for default values, list options.

Features

Python 3.6+ only.

The future is now.

Automatic git repo creation

Adds a github remote, if you like.

Sensible project structure

- pip-installable immediately.
- Modern `setup.py` with `entry_points`

Features

Package management with `pipenv`

- todo: or `venv` or `poetry`
- PRs welcome! :)

Select from standard scientific stack packages per package.

You always get `numpy`, `attrs`, `versioneer`, and some friends.

Test framework and example tests

- tests pregenerated

Command-line interface with `click` or `docopt`

Features

Preconfigured provenance-minded functions.

- `get_configuration`

Automatic versioning

- `versioneer` preinstalled and configured.

Prepopulated README

Installation and Use

Install `cookiecutter` from `pip` or your favorite package manager.

Then,

```
cookiecutter gh:gvoysey/cookiecutter-python-scientific.git -o .
```

Follow the prompts.

Get a cup of coffee.

Enjoy your new project.

Demo time!

Let's see this in action.