Best Practices for Reproducible Research Using Python

Thamme ‘TG’ Gowda
@thammegowda

Click here for Google Slides Link

Thanks: Joel Mathew

9/19/2019
ME = ∫ 2008 to 2019 dt

B.E. in Computer Science and Engineering
@ VTU, Belgaum, India (2008-2012)

M.S. in Computer Science
@ USC Viterbi School of Engineering (2015-2017)

Ph. D. in Computer Science (In Progress)
@ USC Viterbi School of Engineering (2018-*)

Software Engineer
@ SimplyPhi, Bengaluru, India (2012-2015)

Data Scientist (Internship x 3 times)
@ NASA Jet Propulsion Lab (2016-2017)

Research Programmer (Current Role)
@ USC Information Sciences Institute (2017-*)

Technical Co-Founder
@ Datoin, Bengaluru, India (2013-*)

(Volunteer) Committer + PMC Member
@ Apache Software Foundation (2015-2017)

Research Assistant
@ USC Info Retrieval and Data Science (2016)

Mentor
@ Google Summer of Code (2017)

Deep Learning + NLProc research

exploratory + full stack!

Data Science

Big Data + ML + Text Analytics...
Platform building.

@thammegowda
Overview

- Motivation
- Tools and Best Practices
- Portability and Reproducibility
- Readability of Python code
- Some more tools for productivity
Motivation

• Tools and Practices that improve
  - Your Productivity
  - Your code’s Portability
  - Your code’s Readability

1. Productivity FTW!
   a. Collaboration is necessary.
2. Readability and Portability
   a. For successful collaborations
   b. For the *Pride of Workmanship, Satisfaction*
   c. Karma: What you give, comes back to you

*Don’t worry, we are not going to ISO-9126 today!*
Hi! I like to collaborate with you on that awesome project!

I am excited to try some new ideas along your research direction.

Great! First, try to reproduce the results.

I got the results (that was not easy 😞) Time to try my ideas 😊

Sure, please look inside the code; Sorry, not much documentation, I had spent most time on revising paper, had very little for the code.

Good luck; Let me know.
Benefits for You: Productivity

• What? Do more with less time.
• Why? Don’t ruin after-hours, weekends, and sleep
• How? Use right/best tools and practices. They help:
  – Get tasks done faster and much faster: automate
  – Catch bugs ahead of time: have fewer bugs
  – Collaborate: others can help you, only if it’s easy to step-in
  – Organize: make code easy to find and modify

• Which tools and what practices, precisely?
  – Some of them will be covered in this talk
  – Maybe not covered entirely, they will be just pointed out
Benefits for Others:

Portability
Facilitate your peers to easily run your code. As a black-box, without having to look inside.
⇒ Reproducibility

Readability
Use a (code) writing style that is easier for you and your peers to read and understand, without having to pull hairs out.
⇒ Collaboration

How to:
1. Use right tools
2. Use best practices
• Still using Python 2.7?
  Please upgrade to 3.7+

https://xkcd.com/353/
Python

- Portable code - for reproducibility
  - Python is portable, by default
  - Yet we come across code that is so hard to run

- Readable code - for collaboration
  - Python is one of the easiest languages ever (executable-pseudocode)
  - Yet we see cryptic, awkwardish, complicated code
Portable code for Reproducibility

https://xkcd.com/1742/
Setting up a Project

- Create a git repository
  - Need version control/backup: *use git*
  - Multiple features/ideas/fixes in parallel: *use git*
  - Multiple people contribute code in parallel: *use git*
  - ...

- **GitHub** is a goto place for hosting git repos
  - *GitLab is popular too*

- Many useful tools to improve productivity
  - Issues and Discussion threads
  - Pull Requests and Code Reviews
  - Wikis
GitHub Workflow

- Repository
- Merge
- Pull Request + Code Review
- Iterate
- Issue
- Report
- Create
- Fork / Branch
- Edit + Commit
Checklist

✅ Create Github/Gitlab account (if you don’t already have one)
✅ Create a repository for your project. Decide private / public
✅ Add collaborators
✅ Create a README file  

(more details on this soon)
✅ git clone  
✅ git pull
✅ git add  
✅ git commit
✅ git push
✅ git branch  
✅ git checkout
✅ Github Pull Request  
✅ Code Reviews
## Git commit messages in long run

<table>
<thead>
<tr>
<th>Comment</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Created main loop &amp; timing control</td>
<td>14 hours ago</td>
</tr>
<tr>
<td>Enabled config file parsing</td>
<td>9 hours ago</td>
</tr>
<tr>
<td>Misc bugfixes</td>
<td>5 hours ago</td>
</tr>
<tr>
<td>Code additions/edits</td>
<td>4 hours ago</td>
</tr>
<tr>
<td>More code</td>
<td>4 hours ago</td>
</tr>
<tr>
<td>Here have code</td>
<td>4 hours ago</td>
</tr>
<tr>
<td>AAAAAAAA</td>
<td>4 hours ago</td>
</tr>
<tr>
<td>ADKFJ5LKDFJSDFKJF</td>
<td>3 hours ago</td>
</tr>
<tr>
<td>My hands are typing words</td>
<td>2 hours ago</td>
</tr>
<tr>
<td>HAAAAAAAAANDS</td>
<td>2 hours ago</td>
</tr>
</tbody>
</table>

As a project drags on, my Git commit messages get less and less informative.

[https://xkcd.com/1296/](https://xkcd.com/1296/)
git DOs and DON’Ts

✅ DO: Write meaningful and truthful commit messages
✅ DO: Use branches for working in parallel
✅ DO: Always keep the branches up-to date synchronized
✅ DO: Keep master branch in working condition
✅ DO: One commit per sub task: [[Small —<balance>— Big]]

⛔ DON’T: commit generated files
  - such as compiler generated, outputs, and log files
  - binary files that change often
⛔ DON’T: commit unwanted files
⛔ DON’T: commit a huge batch of changes at once
Write a README

- Description: what is this code for?
- Markdown or richer format; sections with headings
- How to install?
- Where are the settings? Incase we need to change any
- A quick example of how to run/use the code is must
- A detailed tutorial will be nicer
- License → Should have a separate discussion on this topic
- Contributors and Acknowledgement
- How to report issues? → use Github/Gitlab issues
Installing Dependencies

- brew apt yum conda pip .... or give a docker image
- Recommend: cross-platform tools: pip and conda
  - 👍 pip and conda work together; you need both
pip

- [https://pypi.org/](https://pypi.org/)
- `pip install <name>`
- `pip install <name>==<version>`
- List down all the libs and versions in `requirements.txt`
  - One `<name>==<version>` per line
- `pip install -r /path/to/requirements.txt`
- 👉🏼 DON’T forget the version numbers
conda

- Download and install miniconda
  https://docs.conda.io/en/latest/miniconda.html
- Suggestion: one conda environment per project
- conda create -n <myenv> python=3.7
- conda activate <myenv>
- conda env create -f environment.yml

- conda can do more than managing python environments
  - It can install system libraries without needing sudo
Likelihood you will get code working based on how you’re supposed to install it:

Very likely
- App Store
- Or package manager
- GitHub link
- SourceForge link
- Geocities/Tripod link
- Copy-and-paste example code from paper’s appendix

Anything that “requires only minimal configuration and tweaking”

Unlikely
Code as a Package

- Create a `setup.py`, with requirements
- It’s easy! Copy-paste a template and modify
- Installation: `pip install` .
  - Development installation: `pip install --editable` .
- Ready to give it to the world for free, release to PyPI
- If you don’t want to `pip`, create a `setup.sh` script
- If data needs to be downloaded, write `get-data.sh` script
from setuptools import setup
from pathlib import Path
import awkg # import own package

long_description = Path('README.md').read_text(encoding='utf-8')
setup(name='awkg', version=awkg.__version__,
      packages=['awkg'],  # for a single .py file, use py_modules=[]
      description=awkg.__description__, long_description=long_description,
      long_description_content_type='text/markdown',
      license='GNU General Public License v3 (GPLv3)',
      classifiers=classifiers, python_requires='>=3.6',
      url='https://github.com/thammegowda/awkg',
      platforms=['any'],
      author='Thamme Gowda',
      entry_points={'console_scripts': ['awkg=awkg:AWKG.main'],})
DON’T write hard local paths

- DONT: hard code local paths
- DO: Use an environment variable
- DO: Make everything relative to it

Example:

$<project>_HOME/data
$<project>_HOME/conf
$<project>_HOME/bin
$<project>_HOME/libs

Image Credit: Reddit
All Configs at One Place

• DON’T spread the configs all over your project code
• DO keep all configs at one place.
• DO create a config for experiment for reproducibility

Format of config file:
• config.py
• config.ini
• config.xml : old school! hard to read/manipulate in python 😞
• config.json : almost usable, but doesn’t support comments
• config.yml : 👍 Use ruamel.yaml to preserve comments
• config.jsonnet : https://jsonnet.org/
Good Use of Existing Env. Variables

- $HOME$ variable
- What if commands were already in PATH?
  - No need to set full path to the command binary
- What if the python code was already in PYTHONPATH?
  - No need for set full path
  - just “from my_script import my_func”
- conda environment can do that for you!
- Try not to invent too many new variables
Improving Readability of Python Code
Follow Python Conventions

- Python community didn’t start with a set of conventions. Developers used whatever conventions they liked.
- No conventions were also okay.
- Conventions have evolved, and became PEP8 or PEP-0008: [https://www.python.org/dev/peps/pep-0008/](https://www.python.org/dev/peps/pep-0008/)
- Use an IDE: such as pycharm
  - Watch out the red and yellow lines
PEP8: Naming Conventions

- ClassName
- method_name() not dontUseMixedCase()
- variable_name not dontUseMixedCase
- _internals_one_underscore
- __two_underscores__ such as __init__()
- CONSTANTS_ARE_CAPS
- dontUseMixedCase, unless already used and it’s too late
- Dont_Do_This_Either

Advantages?
**docstrings and comments**

- **DO:** add docstrings, at least to the public functions
- **Example:**

```python
def manual_seed(seed):
    r"""Sets the seed for generating random numbers. Returns a
    `torch.Generator` object.
    """

    Args:
    
    seed (int): The desired seed.
```

---

*Information Sciences Institute*
Caution: Complexity Increases Over Time

If the code becomes too complex over the time, please refactor code

- Line length: Used to be 80; Can go upto 120 chars now
- Number of lines in function: [Not too many]
- How many arguments to functions: [Not too many]
- How many code files in a directory: [Not too many]
  - Use namespaces/packages: and of course use meaningful names
- Too much Dead Code? Consider removing it!
  - Dead code: commented out source code
  - Don’t worry, git has everything saved for you (if you had committed it)
CLI with argparse

DONT: Directly manipulate sys.argv
foo = sys.argv[1]
bar = sys.argv[2]

DO: Use argparse

parser = argparse.ArgumentParser(description='Description of your program')
parser.add_argument('-f', '--foo', help='Description for foo argument', required=True)
parser.add_argument('-b', '--bar', help='Description for bar argument', required=True)
args = vars(parser.parse_args())
Integrations via subprocess?

- DON’T write everything under `__main__` block
  - Only luck we have with this is call via `subprocess`
  - Often no need for launching frequent external processes
- Setup `PYTHONPATH` properly,
  "from myscript import method"; call "method(args)"
- You can pass complex data structures in memory
- It’s nicer that way than subprocess
  - No unnecessary work like writing and reading files
  - No unnecessary CLI arg parsing and disk IO
Are too many args bad? Example

Example from tensorflow/tensor2tensor:

```python
y = common_attention.multihead_attention(
    common_layers.layer_preprocess(
        x, hparams, layer_collection=layer_collection),
    None,
    decoder_self_attention_bias,
    hparams.attention_key_channels or hparams.hidden_size,
    hparams.attention_value_channels or hparams.hidden_size,
    hparams.hidden_size,
    hparams.num_heads,
    hparams.attention_dropout,
    attention_type=hparams.self_attention_type,
    max_relative_position=hparams.max_relative_position,
    heads_share_relative_embedding=(
        hparams.heads_share_relative_embedding),
    add_relative_to_values=hparams.add_relative_to_values,
    save_weights_to=save_weights_to,
    cache=layer_cache,
    make_image_summary=make_image_summary,
    dropout_broadcast_dims=attention_dropout_broadcast_dims,
    max_length=hparams.get("max_length"),
    decode_loop_step=decode_loop_step,
    vars_3d=hparams.get("attention_variables_3d"),
    activation_dtype=hparams.get("activation_dtype", "float32"),
    weight_dtype=hparams.get("weight_dtype", "float32"),
    layer_collection=layer_collection,
    recurrent_memory=recurrent_memory,
    chunk_number=chunk_number,
    hard_attention_k=hparams.get("hard_attention_k", 0),
    gumbel_noise_weight=hparams.get("gumbel_noise_weight", 0.0),
    max_area_width=max_area_width,
    max_area_height=max_area_height,
    memory_height=memory_height,
    area_key_mode=hparams.get("area_key_mode", "none"),
    area_value_mode=hparams.get("area_value_mode", "none"),
    training=(hparams.get("mode",
        tf.estimator.ModeKeys.TRAIN)
    ==tf.estimator.ModeKeys.TRAIN))
```
class MultiHeadedAttention(nn.Module):
    def __init__(self, n_heads, hid_size, dropout=0.1):
        ...
    def forward(self, query, key, value, mask=None):
        ...

Usage:

multi_attn = MultiHeadedAttention(n_heads, hid_size, dropout=dropout)
attn_val = multi_attn(query, key, value, mask)
Use Logger

Use logger with proper levels

```python
import logging as log
log.basicConfig(level=log.INFO)

log.debug("Building Index...")
log.info("Index is valid")

log.warning("Index is invalid")
log.error("Index is invalid; exiting")
```
New Features

- typing: 3.5+
- f-strings aka literal strings: 3.6+
- pathlib: 3.4+
- dataclasses: 3.7+
typing

- Typed code is easier to understand and debug than non typed
- DO: Annotate public function args with types

def word_count(input):
    # bad arg name; what is this input thing? too broad

def word_count(sentences):
    # good argument name, but how do sentences?

from typing import List, Dict
def word_count(sentences: List[List[str]]) -> Dict[str, int]:
    # Nice huh?
BEFORE

def manual_seed(seed):
    r"""Sets the seed for generating random numbers. Returns a
    `torch.Generator` object.
    Args:
        seed (int): The desired seed.
    """

AFTER

def manual_seed(seed: int) -> torch.Generator:
    r"""Sets the seed for generating random numbers.
    Args:
        seed: The desired seed.
    """
Useful tools and libs
jq

- XML? Use JSON
  - `json.load(...)` and `json.dump(...)`
- Too many JSON Documents? Use JSONLines
  - [http://jsonlines.org/](http://jsonlines.org/)
- `jq` is awesome [https://stedolan.github.io/jq/](https://stedolan.github.io/jq/)
Text Editor vs Notebook vs IDE

- Text Editors: vim/emacs/sublime/atom/brackets ...
  - vim/emacs for tweaking on remote servers via ssh
- Prototype: Jupyter lab (jupyter notebook)
  - pip install jupyterlab
  - Google Colab: https://colab.research.google.com
- Production: Use an IDE
  - PyCharm is awesome https://www.jetbrains.com/pycharm/
  - Pay attention to yellow and red underlines marked by your IDE
We plot the number of passengers at the Rosengartenstrasse stop.

In [93]:
load = df[df.stopNameShort=='ROSE'].passengerLoadStop
sns.distplot(load, kde=False)
plt.axvline(load.median(),)
plt.title('Passenger Load at Rosengartenstrasse stop')
plt.xlabel('Number of passengers'); plt.ylabel('Frequency');
More python libs

- numpy and matplotlib
- pandas
- ML modeling:
  - Pytorch
  - Tensorflow 2.0 with Keras
  - sklearn
- HTTP / REST API:
  - client: requests
  - server: flask
- Web data:
  - XPATH (lxml)
  - scrapy

Try not to reinvent these libs; instead take full advantage.
CLI Tools

Don’t reinvent these:

• grep
• sed
• cut; paste
• awk
• sort; uniq
• jq; yq

Don’t reinvent these, seriously:

• parallel
• rsync
• ssh
Discussion / Thank You