Good Coding Practices

Daniel Perrefort – University of Pittsburgh

What is a Best Practice?

- *Best practices* are any procedure that is accepted as being the most effective either by **consensus** or by **prescription**.
- *Practices* can range from stylistic to in-depth design methodologies.

"A universal convention supplies all of maintainability, clarity, consistency, and a foundation for good programming habits too."

—Tim Peters on comp.lang.python, 2001-06-16

A Roadmap

PEP’s and good styling

Writing good documentation

How to organize your project

Python Enhancement Protocol (PEP)

"A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment." (PEP 1)

Important Fundamentals	**PEP 8:** Style Guide for Python Code
	PEP 20: The Zen of Python
	PEP 257: Docstring Conventions
Bonus PEPs	**PEP 484:** Type Hints
	PEP 498: Literal String Interpolation
	PEP 572: Assignment Expressions

PEP 20: The Zen of Python

```
>>> import this
```

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

PEP 8: Style Guide for Python Code

Code is read much more often than it is written.

Easy to read means easier to develop.

Well written code conveys professionalism.

Code is a deliverable part of your project!

PEP8: Code Layout

Your probably already familiar with…

- Using 4 spaces per indentation level (not tabs!)
- Putting two blank lines before functions and classes
- Limiting line lengths to:
  - 79 characters for code
  - 72 characters for long blocks of text
  - It is okay to increase the line length limit up to 99 characters

PEP8: Code Layout

```
# !/usr/bin/env python3.7
# -*- coding: UTF-8 -*-

"""This is a description of the module."""

import json
import os

from astropy.table import Table, vstack

from my_code import utils

__version__ = 0.1


def my_function():
    ...
```

PEP 8: Naming Conventions

TYPE	NAMING CONVENTION	EXAMPLES
Function	Use lowercase words separated by underscores.	function, my_function
Variable	Use lowercase letters or word, or words separated with underscores. (I.e., snake_case)	x, var, my_variable
Class	Start each word with a capital letter. Do not separate words with underscores. (I.e., CamalCase)	Model, MyClass
Method	Use lowercase words separated with underscores.	class_method, method
Constant	Use an uppercase single letter, word, or words separated by underscores.	CONSTANT, MY_CONSTANT
Module	Use short lowercase words separated with underscores.	module.py, my_module.py
Package	Use short lowercase words without underscores.	package, mypackage

PEP8: General Recommendations

- Use `is` when comparing singletons
- Use `is not` instead of `not ... is`

```
# Wrong
if foo == None:
    do_something()


# Also wrong
if not foo is None:
    do_something()
```

```
# Correct
if foo is not None:
    do_something()
```

PEP8: General Recommendations

- Always use a `def` statement instead of an assignment statement for anonymous (lambda) expressions

```
# Wrong
f = lambda x: 2 * x
```

```
# Correct
def double(x):
    return 2 * x
```

PEP8: General Recommendations

- Derive exceptions from `Exception` rather than `BaseException`
- Use explicit exception catching (avoid bare exceptions)
- Keep `try` statements as simple as possible

```
# Wrong
try:
    import platform_specific_module
    my_function()

except:
    platform_specific_module = None
```

```
# Correct
try:
    import platform_specific_module

except ImportError:
    platform_specific_module = None

else:
    my_function()
```

PEP8: General Recommendations

- Booleans are already Booleans – they don't need comparisons
- For sequences, (e.g., a lists), use the fact that empty sequences are false

```
# Wrong:
if my_boolean == True:
    do_something()

# Worse:
if my_boolean is True:
    do_something()

# Still bad:
if len(my_list) != 0:
    do_something()
```

```
# Correct for sequences and booleans
if some_variable:
    do_something()
```

If You Take Away One Thing...

- PEP8 inspection is built into many Integrated Development Environments (IDEs)
  - **PyCharm**: https://www.jetbrains.com/pycharm/
  - **Atom-pep8**: https://atom.io/packages/pep8
- Command line tools for PEP 8 are also available
  - **Pylint**: http://pylint.pycqa.org/
  - **Flake8**: https://flake8.pycqa.org/
- Jupyter Plugins:
  - **Python Black**: https://github.com/drillan/jupyter-black

Side Note

PEP 257: Docstring Conventions

Documentation is key to reusable code

Never assume you will remember what your code does (or how it works)

Documentation can include technical notes and derivations.

Saves you headaches when you revisit a project in part or in whole

Good Documentation Should…

- Explain what each function / module / package does or is responsible for
- Be understandable to you when you revisit the code in 6 months
- Be understandable by someone new to the project (but not *necessarily* new to the subject matter)
- Be specific and to the point

PEP257: Single Line Docs

- Triple quotes are **always** used
- The closing quotes are on the same line as the opening quotes.
- The docstring is a phrase ending in a period. It prescribes the function's effect as a command ("Do this", "Return that"), not as a description; e.g. don't write "Returns the pathname .

```
def kos_root():
    """Return the pathname of the KOS root directory."""

    ...
```

PEP257: Multi-Line Docs

- Start with a single line docstring
- Include additional documentation as necessary
- Always document arguments and returns

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Here is where you would put some additional, in-depth documentation.

    Keyword arguments:
        real -- the real part (default 0.0)
        imag -- the imaginary part (default 0.0)

    Returns:
        An imaginary number corresponding to the given arguments
"""
```

External Style Guides (Google)

- Based on the principle that docs in the code should be human readable

```
def connect_to_next_port(minimum):
    """Connects to the next available port.

    Args:
        minimum: A port value greater or equal to 1024.

    Returns:
        The new minimum port.

    Raises:
        ConnectionError: If no available port is found.
    """
```

http://google.github.io/styleguide/pyguide.html

Document the Code **AND** the Project

- Extensive project documentation isn't always necessary and should scale to meet your project requirements.

- Include a README file at minimum
  - Describe the project goals and general approach
  - Does not need to be super in depth

- For larger projects, you might document:
  - Design choices or style guides
  - Project notes (e.g. from papers you read)
  - A development plan / roadmap

Side Note

- Use tools like **Sphinx** and **Read The Docs** to generate automatic documentation
  - Sphinx: https://www.sphinx-doc.org/
  - RTD: https://readthedocs.org
- Running the setup script:

```
$ pip install sphinx
$ sphinx-quickstart
```

PEP 484: Type Hints

- "New" as of Python 3.5
- Not extensively used but can be extremely helpful for
  - Adding inspection support when developing API's
  - Enforcing type linting in your own projects

Type hints are probably not a "best practice" but planning out your code ahead of time (e.g. function signatures) is!

```
from typing import Union

PathLike = Union[str, Path]


def greeting(name: str) -> str:
    return 'Hello ' + name


def process_directory(path: PathLike):
    return 'Hello ' + name
```

How to Organize Your Project

Proper organization promotes reproducibility

How you set up your project effects your ability to collaborate

Version control provides continuity and collaboration

Virtual environments eliminate dependency conflicts

Source Code Organization: Directories

DIRECTORY	USAGE
source	Your project source code. The code responsible for **performing** your analysis.
scripts	Individual scripts responsible for **running** separate stages of your analysis.
plotting	Scripts for creating finalized plots.
docs	Stores your project's documentation.
notebooks	For holding notebooks used in exploratory analysis.
tests	Your project test suite.
examples	Use if you want to demonstrate your project.

FILE	USAGE
README.md	Provides a project description.
requirements.txt	Outlines your project dependencies.
LICENSE.txt	License for the distribution of your code (or the forked source). (GNU)

The Infamous “Scripts” Directory

- Scripts should **NOT** be where your analysis logic is
- Scripts should **NOT** be a dumping ground for scratch code
- Each script should represent a single distinct task. For e.g.,
  - Run image calibration
  - Fit object light-curves
  - Download / format data from a remote server
- Include (short) module level docs for each script

Use Version Control

- Allows easier collaboration, especially with large teams.
- Provides descriptions of each change and why it was made.
- Backs up your project incase something goes wrong.
- You can revert changes or recover previous code.

Git Cheat Sheet

GIT BASICS

`git init <directory>`	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
`git clone <repo>`	Clone repo located at *<repo>* onto local machine. Original repo can be located on the local filesystem or on a remote machine via *HTTP* or *SSH*.
`git config user.name <name>`	Define author name to be used for all commits in current repo. Devs commonly use `--global` flag to set config options for current user.
`git add <directory>`	Stage all changes in *<directory>* for the next commit. Replace *<directory>* with a *<file>* to change a specific file.
`git commit -m "<message>"`	Commit the staged snapshot, but instead of launching a text editor, use *<message>* as the commit message.
`git status`	List which files are staged, unstaged, and untracked.
`git log`	Display the entire commit history using the default format. For customization see additional options.
`git diff`	Show unstaged changes between your index and working directory.

UNDOING CHANGES

`git revert <commit>`	Create new commit that undoes all of the changes made in *<commit>*, then apply it to the current branch.
`git reset <file>`	Remove *<file>* from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
`git clean -n`	Shows which files would be removed from working directory. Use the *-f* flag in place of the *-n* flag to execute the clean.

REWRITING GIT HISTORY

`git commit --amend`	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
`git rebase <base>`	Rebase the current branch onto *<base>*. *<base>* can be a commit ID, branch name, a tag, or a relative reference to *HEAD*.
`git reflog`	Show a log of changes to the local repository's *HEAD*. Add `--relative-date` flag to show date info or `--all` to show all refs.

GIT BRANCHES

`git branch`	List all of the branches in your repo. Add a *<branch>* argument to create a new branch with the name *<branch>*.
`git checkout -b <branch>`	Create and check out a new branch named *<branch>*. Drop the *-b* flag to checkout an existing branch.
`git merge <branch>`	Merge *<branch>* into the current branch.

REMOTE REPOSITORIES

`git remote add <name> <url>`	Create a new connection to a remote repo. After adding a remote, you can use *<name>* as a shortcut for *<url>* in other commands.
`git fetch <remote> <branch>`	Fetches a specific *<branch>*, from the repo. Leave off *<branch>* to fetch all remote refs.
`git pull <remote>`	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
`git push <remote> <branch>`	Push the branch to *<remote>*, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

ATLASSIAN

Visit atlassian.com/git for more information, training, and tutorials

Put one of these at your desk! (atlassian)

Virtual Environments

- Use a different environment for each project
- Prevents dependency conflicts and encapsulates projects separately.
- Environments can be shared!

```
$ conda create –n my_environment python=3.8
$ conda activate my_environment
$ ...

$ conda deactivate
```

Conclusions

- Focus on clean, organized code
  - Easier to develop and collaborate on
  - Conveys professionalism
- Always include documentation for your code
  - Scale to project needs
- Keep your projects organized for reproducibility