Python code design and documentation

Increasing the usability of your code

Tristan Kuehn

Intro

- Once you know enough Python to get what you need done, it can be tricky to know how to improve your code
- Ideally, your code will be readable, correct, and usable.
- When possible, it’s a good practice to share any analysis code alongside a paper
  - You get the most value out of this practice when others can understand and use the code!
- There are lots of concrete tools and techniques you can use here
- To a less experienced user, it can be hard to find and apply them
- This will be a survey of these tools and techniques at a high level
- I won’t get into installation, but most IDEs will have some support built in.

Readability

Can you understand what this code is doing?

General points on readability

- It takes a lot longer and is harder to work with code that’s difficult to understand
- As you’re writing your code you know it better than anyone
  - Don’t just assume it’s easy to understand because of this!
  - Put yourself in a naive user’s shoes, or ask a colleague to take a look at it.
- If you come back to some code you’ve written after a couple of months or longer, you are a naive user again!
- This is an art more than a science, but there are some tools and heuristics you can use for help

Names

- Descriptive names go a long way toward making your code readable
- Some tension between clarity and length
  - `n` is usually not a good name, but neither is `number_of_voxels_in_my_region_of_interest`
- Python guidelines for case: PEP 8
- Functions/methods should be verbs in imperative voice
  - `print`, not `printer` or `prints`
- This can be hard!
  - “There are only two hard things in Computer Science: cache invalidation and **naming things**.”

Code formatters

- It’s a huge pain to manually keep up consistent formatting in a project.
- How to split long lines in different situations? Single quotes or double quotes? How many blank lines? Where to put parentheses?
- A code formatter handles all these concerns automatically.
- Popular examples: black, autopep8
- Well-integrated in IDEs, can be configured to run automatically or on demand

Black example

```
if very_long_variable_name is not None and \
 very_long_variable_name.field > 0 or \
 very_long_variable_name.is_debug:
 z = 'hello '+'world'
else:
 world = 'world'
 a = 'hello {}'.format(world)
 f = rf'hello {world}'
if (this
and that): y = 'hello ''world'#FIXME: https://github.com/psf/black/issues/26
class Foo  (     object  ):
  def f    (self    ):
    return       37*-2
  def g(self, x,y=42):
      return y
def f  (   a: List[ int ]) :
  return      37-a[42-u :  y**3]
def very_important_function(template: str,*variables,file: os.PathLike,debug:bool=False,):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
     ...
```

Black example

```python
if (
    very_long_variable_name is not None
    and very_long_variable_name.field > 0
    or very_long_variable_name.is_debug
):
    z = "hello " + "world"
else:
    world = "world"
    a = "hello {}".format(world)
    f = rf"hello {world}"
if this and that:
    y = "hello " "world"  # FIXME: https://github.com/psf/black/issues/26


class Foo(object):
    def f(self):
        return 37 * -2

    def g(self, x, y=42):
        return y


def f(a: List[int]):
    return 37 - a[42 - u : y**3]


def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    debug: bool = False,
):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
        ...
```

Refactoring into functions

- If all your code is written exclusively as executable scripts, it's hard to reuse it and often hard to make changes or maintain it.
- Instead, it's helpful to:
  - Break your code into logical units
    - The interior of loops if they start to get long
    - Any block of code that's copy/pasted (or very similar) in multiple parts of a script
    - Complicated boolean expressions
  - Write those units as more general functions
  - Write a "main" function that calls your functions in order
    - `if __name__ == "__main__":` pattern is useful
- Linters can pick up complex parts of code and recommend a refactor
  - More on this later.

Refactoring example

```
def pix_collection(items):
    res = []
    limit = 10
    for i in items:
        if (i[0] ** 2 + i[1] ** 2) < limit**2:
            res.append(i)
    return res
```

Adapted from Serge Koudoro. “Secret Session: Master coding in your research environment. Code Documentation.” ISMRM 2019.

Refactoring example

```python
def pix_collection(items):
    res = []
    limit = 10
    for i in items:
        if (i[0] ** 2 + i[1] ** 2) < limit**2:
            res.append(i)
    return res
```

```python
def is_in_origin_circle(x_coord, y_coord, limit):
    return (x_coord**2 + y_coord**2) < limit**2


def filter_neighbourhood_pixels(pixels, limit=10):
    neighbourhood_pixels = []
    for pixel in pixels:
        if is_in_origin_circle(pixel[0], pixel[1], limit):
            neighbourhood_pixels.append(pixel)
    return neighbourhood_pixels
```

- Factor out distance predicate
- Rename almost everything
- Make `limit` an optional argument

Refactoring example

```python
def pix_collection(items):
    res = []
    limit = 10
    for i in items:
        if (i[0] ** 2 + i[1] ** 2) < limit**2:
            res.append(i)
    return res
```

```python
def is_in_origin_circle(x_coord, y_coord, limit):
    return (x_coord**2 + y_coord**2) < limit**2


def filter_neighbourhood_pixels(pixels, limit=10):
    neighbourhood_pixels = []
    for pixel in pixels:
        if is_in_origin_circle(pixel[0], pixel[1], limit):
            neighbourhood_pixels.append(pixel)
    return neighbourhood_pixels
```

```python
def is_in_origin_circle(x_coord, y_coord, limit):
    return (x_coord**2 + y_coord**2) < limit**2


def filter_neighbourhood_pixels(pixels, limit=10):
    return [pixel for pixel in pixels if is_in_origin_circle(*pixel, limit)]
```

- Use a list comprehension
- Unpack `pixels` directly as an argument

Commenting

- Comments are generally a good thing, but don’t go overboard
- Exception: docstrings for modules, functions, and classes
  - IDEs, Autogenerated documentation (Sphinx autodoc), `__doc__`
- Otherwise comments are exclusively for when you’ve done something non-obvious
- Ask yourself if you can make the code clearer before writing a comment
  - Change some names
  - Refactor
  - Add type hints (more on this later)
- If you have to do something weird, write a concise comment explaining it and include any context.

Correctness

Does your code do what you want it to do?

General points on correctness

- The most important part of any code you right is that it works!
- Most basic way to test this is to run it (with some real input data) and manually inspect the results.
- This can be hard to do with bigger projects.
- There are tools that try to identify problems before you run some code
- There are also tools that automatically test parts of your code
- Both can be useful

Linters

- Linter: Tool that analyzes code and makes suggestions automatically
- Pylint is the big one for Python
- For the purposes of code correctness, pay close attention to E- (error) and W- (warning) level messages.
- This can help catch everything from simple typos to subtle Python errors before you try to use your code.
- Pylint is available in every major IDE I’m aware of (VSCode, Spyder, …) or just from the command line.

Pylint example

```python
def print_strings(strings=[]):
    """Print a list of strings plus a default string on the end"""
    strings.append("always_printed")
    print(f"Printing {len(strings)} strings")
    for str_ in strings:
        print(str_)
    return
    print(f"Printed {len(strings)} strings.")

if __name__ == "__main__":
    print_strings()
    print_strings(["hi"])
    print_strings()
```

```
Printing 1 strings
always_printed
Printing 2 strings
hi
always_printed
Printing 2 strings
always_printed
always_printed
```

Pylint example

```python
def print_strings(strings=[]):
    """Print a list of strings plus a default string on the end"""
    strings.append("always_printed")
    print(f"Printing {len(strings)} strings")
    for str_ in strings:
        print(str_)
    return
    print(f"Printed {len(strings)} strings.")

if __name__ == "__main__":
    print_strings()
    print_strings(["hi"])
    print_strings()
```



```
Printing 1 strings
always_printed
Printing 2 strings
hi
always_printed
Printing 2 strings
always_printed
always_printed
```

```
linttest.py:1:0: C0114: Missing module docstring (missing-module-docstring)
linttest.py:1:0: W0102: Dangerous default value [] as argument (dangerous-defau
lt-value)
linttest.py:8:4: W0101: Unreachable code (unreachable)
```

Pylint example

```python
def print_strings(strings=[]):
    """Print a list of strings plus a default string on the end"""
    strings.append("always_printed")
    print(f"Printing {len(strings)} strings")
    for str_ in strings:
        print(str_)
    return
    print(f"Printed {len(strings)} strings.")

if __name__ == "__main__":
    print_strings()
    print_strings(["hi"])
    print_strings()
```



```
Printing 1 strings
always_printed
Printing 2 strings
hi
always_printed
Printing 2 strings
always_printed
always_printed
```

```python
DEFAULT_STRING = "always_printed"

def print_strings(strings=None):
    """Print a list of strings plus a default string on the end"""
    to_print = (strings + [DEFAULT_STRING]) if strings else [DEFAULT_STRING]
    print(f"Printing {len(to_print)} strings")
    for str_ in to_print:
        print(str_)
    print(f"Printed {len(to_print)} strings.")
    return
```



```
Printing 1 strings
always_printed
Printed 1 strings.
Printing 2 strings
hi
always_printed
Printed 2 strings.
Printing 1 strings
always_printed
Printed 1 strings.
```

Testing

- It’s always a good idea to verify that your code works on a small example.
- In general, try to isolate the parts of your code that surround use of an external package
  - Don’t just write tests verifying that popular packages like numpy work.
- Admittedly this gets a lot harder the more complex your project is, but even putting together one or two end-to-end test cases where you know the expected result is worthwhile.
- Helpful tools: unittest, pytest
- Note: This is a deep topic, we’re barely scratching the surface here

Testing example

From the pytest docs…

Create a new file called `test_sample.py`, containing a function, and a test:

```
# content of test_sample.py
def func(x):
    return x + 1


def test_answer():
    assert func(3) == 5
```

The test

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-7.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_sample.py F                                                  [100%]

================================= FAILURES =================================
_____________________________ test_answer _____________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
======================== short test summary info ========================
FAILED test_sample.py::test_answer - assert 4 == 5
========================== 1 failed in 0.12s ===========================
```

Type checkers

- Automated tools that go a little further to identify errors
- Specifically, ensures that the types of your data are compatible
  - Types: string, int, float, list, …
- Getting the most out of these tools requires type annotations
- e.g. `my_number: float = 6`
- Again, will catch a lot of simple mistakes, but can also catch more subtle errors.
- Examples: pyright, mypy

pyright example

```python
def add_to_one(number_2):
    return 1 + number_2


if __name__ == "__main__":
    print(add_to_one(input("Please enter a number to be added to 1: ")))
```

pyright example

```python
def add_to_one(number_2):
    return 1 + number_2


if __name__ == "__main__":
    print(add_to_one(input("Please enter a number to be added to 1: ")))
```



```
Please enter a number to be added to 1: 2
Traceback (most recent call last):
  File "typetest.py", line 6, in <module>
    print(add_to_one(input("Please enter a number to be added to 1: ")))
  File "typetest.py", line 2, in add_to_one
    return 1 + number_2
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

pyright example

```python
def add_to_one(number_2: float):
    return 1 + number_2


if __name__ == "__main__":
    print(add_to_one(input("Please enter a number to be added to 1: ")))
```



```
pyright 1.1.292
/cifs/khan/users/tkuehn/code/typetest/typetest.py
  /cifs/khan/users/tkuehn/code/typetest/typetest.py:6:22 - error: Argument of type "str" c
annot be assigned to parameter "number_2" of type "float" in function "add_to_one"
    "str" is incompatible with "float" (reportGeneralTypeIssues)
1 error, 0 warnings, 0 informations
```

pyright example

```python
def add_to_one(number_2: float):
    return 1 + number_2


if __name__ == "__main__":
    print(add_to_one(float(input("Please enter a number to be added to 1: "))))
```



```
Please enter a number to be added to 1: 2
3.0
```

Usability

How easy will it be for others (or your future self) to use your code?

Usability

- A lot of time is spent writing code handling a problem that someone else has already addressed
- You can avoid this by sharing your code and making it easy for others to use
- Readability is a big part of this, but at a base level others need to be able to install the dependencies and adapt your script to their data
- We'll talk about some tools that will make that process smoother.

Command line interface

- For tools/scripts, it’s often helpful to provide a command line interface
- Makes it easier to adapt to new data, new environment
- Also makes it easier to bash script with your tool.
- Libraries for this:
  - argparse
  - click

CLI example

```python
"""A script to threshold an image."""

import argparse

import nibabel as nib
import numpy as np
from skimage.filters import threshold_otsu


def threshold_image(image: np.ndarray) -> np.ndarray:
    """Threshold an image using Otsu's method."""
    return image > threshold_otsu(image)


def gen_parser() -> argparse.ArgumentParser:
    """Generate a CLI parser to threshold an image."""
    parser = argparse.ArgumentParser()
    parser.add_argument("image_path")
    parser.add_argument("out_path")
    return parser


def main():
    """Parse an input and output file from command line and threshold the input."""
    parser = gen_parser()
    args = parser.parse_args()
    image = nib.load(args.image_path)
    image_foreground = threshold_image(image.get_fdata())
    nib.save(
        nib.nifti1.Nifti1Image(image_foreground.astype(np.short), image.affine),
        args.out_path,
    )


if __name__ == "__main__":
    main()
```

Dependency Specification

- To use your package, someone needs to know which dependencies they need (numpy, nibabel, scipy are common ones).
- Several ways to do this:
  - requirements.txt: simplest, supported by pip, loosely defined dependencies can cause issues
  - One-liner to generate a requirements.txt: `pip freeze > requirements.txt`
  - You can then trim it down to the necessities
- Installing from a requirements.txt: `pip install -r requirements.txt`
- Even better: Set up a distribution package

Dependencies example – requirements.txt

`pip freeze` output

```
imageio==2.25.0
networkx==3.0
nibabel==5.0.0
numpy==1.24.2
packaging==23.0
Pillow==9.4.0
pkg_resources==0.0.0
PyWavelets==1.4.1
scikit-image==0.19.3
scipy==1.10.0
tifffile==2023.2.3
```

After editing to the essentials

```
nibabel~=5.0
scikit-image~=0.19.3
```

Packaging

- If you're distributing a Python package to multiple people, it can be useful to generate a distribution package for it.
- A distribution package can be pushed to PyPI, making it available via pip's default repository (i.e. `pip install mypackage`)
- Recommended tools:
  - poetry: Newer, checks dependencies to ensure they're internally consistent
  - setuptools: Classic standard, relatively easy to set up.

Packaging example – setuptools

pyproject.toml

```
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"
```

setup.cfg

```
[metadata]
name = myscript
version = 0.0.1

[options]
install-requires =
  nibabel ~=5.0
  scikit-image ~=0.19.3
```

Packaging example – poetry

```
This command will guide you through creating your pyproject.toml config.

Package name [myscript]:
Version [0.1.0]:
Description []:
Author [Tristan Kuehn <tristankuehn@gmail.com>, n to skip]:
License []:
Compatible Python versions [^3.8]:

Would you like to define your main dependencies interactively? (yes/no) [yes]
You can specify a package in the following forms:
  - A single name (requests): this will search for matches on PyPI
  - A name and a constraint (requests@^2.23.0)
  - A git url (git+https://github.com/python-poetry/poetry.git)
  - A git url with a revision (git+https://github.com/python-poetry/poetry.git#develop)
  - A file path (../my-package/my-package.whl)
  - A directory (../my-package/)
  - A url (https://example.com/packages/my-package-0.1.0.tar.gz)

Package to add or search for (leave blank to skip): nibabel
Found 20 packages matching nibabel
Showing the first 10 matches

Enter package # to add, or the complete package name if it is not listed []:
 [ 0] nibabel
 [ 1] nitransforms
 [ 2] indexed-gzip-fileobj-fork-epicfaace
 [ 3] indexed-gzip
 [ 4] cvu
 [ 5] simplebrainviewer
 [ 6] pydeface
 [ 7] morphonet
 [ 8] bidsify
 [ 9] scanphyslog2bids
 [10]
 > 0
Enter the version constraint to require (or leave blank to use the latest version):
Using version ^5.0.0 for nibabel
```

pyproject.toml (not shown: poetry.lock)

```
[tool.poetry]
name = "myscript"
version = "0.1.0"
description = ""
authors = ["Tristan Kuehn <tristankuehn@gmail.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.8"
nibabel = "^5.0.0"
scikit-image = "^0.19.3"

[tool.poetry.group.dev.dependencies]
black = "^23.1.0"
pylint = "^2.16.1"
pyright = "^1.1.292"
flake8 = "^6.0.0"
isort = "^5.12.0"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Wrap-up

- I covered a lot of ground here, so if you’re not using any of these tools/techniques already it would be hard to adopt them all at once
- I do encourage you to pick something that sounded useful and give it a shot, and try incorporating these concepts one-by-one.
- Note: While I was talking about Python, analogous tools and concepts exist for most other mainstream programming languages.

Any questions?