Best Practices

Software Engineering for Scientists

Contents

Follow a Style Guide	2
Document and comment Your Code	3
Use small Functions that do one thing	6
Use the `main` Function unless the file is only a library	8
Use `argparse` to Handle Input Parameters	10
Handle your Exceptions do not pass them to the user and Set Exit Codes	13
Use the clone, Branch, Commit, Push, Pull Request, Merge, tag, Pull Git Workflow	16
Organize Your Repository	18
Write a Good GitHub README	19

Follow a Style Guide

Well formatted code is a requirement of this class and a good practice to maintain in all of your projects. Code is read much more often than it is written, and it is in your interest to make that code as readable as possible. Comments are also important, but they are not a replacement for good code. So that there is no ambiguity about what counts as well-formatted code, we will be using PEP8 (https://www.python.org/dev/peps/pep-0008/ or https://pep8.org/).

PEP8 is a comprehensive guide about how your code should look, and your code should (unless you have a good reason not to) follow these recommendations. The guide is broken up into sections as has many good examples. Such as:

No:

```
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

Yes:

```
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

Check your code with `pycodestyle` install it with `pip` or `conda`

`style_test.py`

```
import sys, so

foo = long_function_name(var_one, var_two,
    var_three, var_four)

a=10
if(a<20): print('small')
```

```
$ pycodestyle style_test.py
style_test.py:1:11: E401 multiple imports on one line
style_test.py:3:43: W291 trailing whitespace
style_test.py:4:5: E128 continuation line under-indented for visual indent
style_test.py:6:2: E225 missing whitespace around operator
style_test.py:7:5: E225 missing whitespace around operator
style_test.py:7:9: E701 multiple statements on one line (colon)
style_test.py:8:1: W391 blank line at end of file
```

Add `--show-source` and/or `--show-pep8` for more output and https://pycodestyle.readthedocs.io for a description of the output and advanced features.

Document and comment Your Code https://realpython.com/documenting-python-code/

The scientific community's adoption of open-source practices has had many benefits for both the users and producers of software, but many other-wise good projects have little impact because of poor documentation. You cannot expect that simply putting your project on GitHub will lead to more users or that your experiments are now reproducible. Your code must include good documentation and comments.

What is the difference? Comments are short statements that are meant for project developers (i.e., you and your lab) to clarify specific code blocks. Documentation is for your users to help them understand functionality so that they may use your code in their research.

Commenting:

Comments being with a `#` and should be brief, free of complex formatting, and as close as possible to the code it is describing as possible. Comments are useful:

- for describing algorithmic or specific behavior of a function choices or the logic behind specific choices

```
# The list is often sorted, so we don't user quick sort
V_sorted = heapsort(V)

# If the file exists values are appended, if not a new file is created
store(file_name, V_sorted)
```

- in the development process to help plan future functionality (`# TODO`) and flag known issues (`# BUG`). These comments should be removed once the issues have been addressed.

```
# TODO: sort this list then do binary search
for i in range(len(V)):
    if V[i] == key:
        hit = i

# BUG: catch the execution for an empty list
mean = sum(V)/len(V)
```

The best way to make your code readable is to design it to comment itself. Use clear variable and function names and and easy-to-understand organization. In many cases we hack until it works, but going back and restructuring so that the logic is clear will avoid the need for excessive comments.

Documenting:

Documenting in Python is done with docstrings, which are simply strategically-placed strings wrapped in triple-double quotes (`"""`) that contain information about the functionality of a class, function, or script. When placed correctly, these strings will be stored in an object's `__doc__` field and can be accessed using python's built-in `help()` function.

For scripts, the docstring comes first, and for functions the docstring immediately follows the `def`.

math_functions.py

```
"""Various math functions

    * get_array_mean - returns the arithmetic mean of a non-empt array
"""

def get_array_mean(V):
    """Compute the arithmetic mean of an array. Expects a non-empty array.

    Parameters
    ----------
    V : list of int
        Non-empty array containing numbers whose mean is desired.

    Returns
    -------
    m
        Arithmetic mean of the values in V

    """
    m = sum(V)/len(V)
    return m
```

```
$ python
>>> import math_functions
>>> help(math_functions)

Help on module math_functions:

NAME
    math_functions - Various math functions

DESCRIPTION
    * get_array_mean - returns the arithmetic mean of a non-empt array

FUNCTIONS
    get_array_mean(V)
        Compute the arithmetic mean of an array. Expects a non-empty array.

        Parameters
        ----------
        V : list orf int
            Non-empty array containing numbers whose mean is desired.

        Returns
        -------
        m
            Arithmetic mean of the values in V

FILE
    /.../math_functions.py
```

```
>>> help(math_functions.get_array_mean)

Help on function get_array_mean in module math_functions:

get_array_mean(V)
    Compute the arithmetic mean of an array. Expects a non-empty array.

    Parameters
    ----------
    V : list orf int
        Non-empty array containing numbers whose mean is desired.

    Returns
    -------
    m
        Arithmetic mean of the values in V
```

There are a number of guides for what to include in a docstring. In this class we will use https://numpydoc.readthedocs.io/en/latest/format.html. There is a lot of information here, but don't feel like you must include every attribute in every case. Focus on the most important details to avoid writer and reader fatigue.

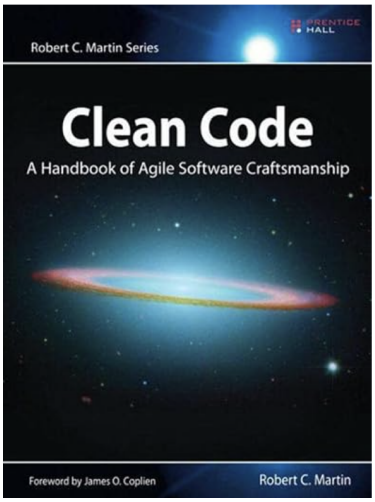
Use small Functions that do one thing

Writing reusable code is fundamental to good software because it improves:

1. **Efficiency**: Once you've written and tested a piece of code, reusing it saves time over writing new, functionally equivalent code from scratch.
2. **Maintainability**: When you need to fix a bug or make an enhancement in a reusable component, you make the change in one place, and all the parts of your application that use that component benefit. This contrasts with having to hunt down and modify every occurrence of duplicated logic.
3. **Reduced Error**: The more you use and reuse a piece of code, the more you test it, and the more likely it is that any bugs in that code will be found and fixed.
4. **Easier Testing**: Reusable components can be tested in isolation, ensuring that they are robust and reliable.
5. **Improved Code Readability**: When code is modular and reusable, it's often clearer and easier to understand, since the functionality encapsulated by reusable components can be understood in isolation.

Writing reusable code means putting your code in functions.

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.



Function rules:

1. **Small**: Long functions are hard to understand and test, and usually break rule 2. How small? 10 lines. Higher-level functions may be longer when they call many smaller functions, read files, etc.
2. **Do one thing**: The most important rule. If your function does two things, you reduce reusability (what if you need to do just the one thing) and make testing hard.
3. **Use descriptive names**: Do not assume that you will remember what you wrote for longer than a few weeks. Pick a name clearly tells you what the function does. Functions that are hard to name probably break rules 1 and 2.
4. **Prefer fewer arguments**: Meh. Assuming your function follows rule 1 and 2, then reading the number of arguments involves stuffing arguments into a container, which has overhead and its own complexities.

```
def foo(too,
        many,
        arguments,
        isnt,
        the,
        worst):
```

```
args = [too, many,
        arguments, isnt,
        the, worst]

def foo(ListsBad):
  too = ListsBad[0]
  many = LitsBad[1]
```

```
args = {'too': 1,
        'many': False,
        ...}

def foo(DictBetter):
  DictBetter['too']
  DictBetter['anythinggoes']
```

```
from collections import namedtuple

Args = namedtuple('Args', ['too',
                           'many',
                           ...])

def foo(NamedTupleEvenBetter):
  NamedTupleEvenBetter.too
```

5. **Have no side effects**: Side effects rely on, or modify, something outside the function signature. Side effects reduce reusability and, more importantly, increase the likelihood of bugs when you inevitably forget about the side effect. Avoid side effects by following rule 2 and returning values instead of modifying variables.

```
a = [1,2,3,4]
def get_mean_with_sideffect(a):
    total = 0
    for i in range(len(a)):
        total += a[i]
    a.append(total/len(a))
get_mean(a)
```

```
a = [1,2,3,4]
def get_mean_without_sideffect(a):
    total = 0
    for i in range(len(a)):
        total += a[i]
    return total/len(a)
mean = get_mean(a)
```

If you MUST have a side effect, document it.

6. **Don't use flag arguments**: Split method into several independent methods that can be called from the client without the flag. Meh.

Use the `main` Function unless the file is only a library

Python is an interpreted language and all of the code in a source file is executed line-by-line when the file is loaded by either direct execution (i.e., `$ python foo.py`) or by importing as a module (i.e, `import foo`), which can have unintended consequences.

Suppose we start with the following source that we execute directly:

`math_lib.py`

```
def list_sum(L):
    s = 0
    for l in L:
        s += l
    return s


def list_product(L):
    s = 1
    for l in L:
        s *= l
    return s


V = [1, 2, 3, 4, 5, 6]

print(list_sum(V))
print(list_product(V))
```

```
$ python math_lib.py
21
720
```

If we then have a new file were we would like to reuse the `list_sum` and `list_prouct` functions

`foo.py`

```
import math_lib


X = [10, 20, 30, 40, 50],

print(math_lib.list_sum(X))
print(math_lib.list_product(X))
```

The execution of this file would also execute any line in `math_lib` that is not in a function

```
$ python foo.py
21
720
150
12000000
```

To prevent these unwanted commands from being executed we can use the `main` function to specify what parts of `math_lib.py` are exected when its a library and when it is called directly

`math_lib.py`

```
def list_sum(L):
    s = 0
    for l in L:
        s += l
    return s


def list_product(L):
    s = 1
    for l in L:
        s *= l
    return s

def main():
    V = [1, 2, 3, 4, 5, 6]

    print(list_sum(V))
    print(list_product(V))

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

```
$ python foo.py
150
12000000
```

Use `argparse` to Handle Input Parameters

Do NOT rely on the user to remember the parameters and their order. By using `argparse` (https://docs.python.org/3/library/argparse.html), every input parameter will be associated with a **reasonably named** command-line argument, the argument order will not matter, and `argparse` will produce a nicely formatted error if there is an issue with the arguments or a helpful message to users when prompted with the `-h` flag.

`the_bad_way.py`

```
import sys

file_name = sys.argv[1]
column_number = int(sys.argv[2])

print(file_name, args.column_number)
```

Unhelpful errors when users:

- Forget the parameters

```
$ python the_bad_way.py
Traceback (most recent call last):
  File "the_bad_way.py", line 3, in <module>
    file_name = sys.argv[1]
IndexError: list index out of range
```

- Get the wrong number of parameters

```
$ python the_bad_way.py test.txt
Traceback (most recent call last):
  File "the_bad_way.py", line 4, in <module>
    column_number = int(sys.argv[2])
IndexError: list index out of range
```

- Or get the order wrong

```
$ python the_bad_way.py 1 test.txt
Traceback (most recent call last):
  File "the_bad_way.py", line 4, in <module>
    column_number = int(sys.argv[2])
ValueError: invalid literal for int() with base 10: 'test.txt'
```

the_good_way.py

```
import argparse

parser = argparse.ArgumentParser(
            description='The right way to pass parameters.',
            prog='the_good_way')

parser.add_argument('--file_name',
                    type=str,
                    help='Name of the file',
                    required=True)

parser.add_argument('--column_number',
                    type=str,
                    help='The column number',
                    required=True)

args = parser.parse_args()

print(args.file_name, args.column_number)
```

- Parameter order doesn't matter

```
$ python the_good_way.py --file_name text.txt --column_number 1
text.txt 1

$ python the_good_way.py --column_number 1 --file_name text.txt
text.txt 1
```

- Helpful error messages if something is missing

```
$ python the_good_way.py
usage: the_good_way [-h] --file_name FILE_NAME --column_number COLUMN_NUMBER
the_good_way: error: the following arguments are required: --file_name,
--column_number

$ python the_good_way.py --column_number 1
usage: the_good_way [-h] --file_name FILE_NAME --column_number COLUMN_NUMBER
the_good_way: error: the following arguments are required: --file_name
```

- Full usage when prompted

```
$  python the_good_way.py -h
usage: the_good_way [-h] --file_name FILE_NAME --column_number COLUMN_NUMBER

The right way to pass parameters.

optional arguments:
  -h, --help            show this help message and exit
  --file_name FILE_NAME
                        Name of the file
  --column_number COLUMN_NUMBER
```

```
                        The column number
```

Handle Your Exceptions do not pass them to the user and Set Exit Codes

Runtime errors in Python are handled using exceptions, which interrupt the normal execution of the program and if they are not handled correctly can terminate the program leaving users with an overly technical and confusing error message.

For example, users will be stuck with the following output if you try and open a file that does not exist:

un_handled_open.py

```
f = open('no_file.txt', 'r')
```

```
$ python un_handled_open.py
Traceback (most recent call last):
  File "un_handled_open.py", line 1, in <module>
    f = open('no_file.txt', 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'no_file.txt'
```

or access an element outside of an array

un_handled_access.py

```
A = [1, 2, 3, 4]

print(A[4])
```

```
$ python un_handled_access.py
Traceback (most recent call last):
  File "un_handled_access.py", line 3, in <module>
    print(A[4])
IndexError: list index out of range
```

There are many built-in exceptions (https://docs.python.org/3/library/exceptions.html) and you can define custom ones by extending the `Exception` class, and a given function can raise different types of exceptions depending on what caused the error.

To catch an exception you must wrap the code in a try-except handler, which can be generic (handle any exception)

handled_access.py

```
A = [1, 2, 3, 4]

try:
    print(A[4])
except:
    print('Something went wrong, not sure what')
```

```
$ python handled_access.py
Something went wrong, not sure what
```

or specific to different exception types

`handled_open.py`

```
def append_to_file(file_name):
    f = None
    try:
        f = open(file_name, 'r')
    except FileNotFoundError:
        print('Could not find ' + file_name)
    except PermissionError:
        print('Could not open ' + file_name)
    finally:
        return f

f = append_to_file('no_file.txt')
f = append_to_file('no_read_permission.txt')
```

```
$ python handled_open.py
Could not find no_file.txt
Could not open no_read_permission.txt
```

Exit codes

When a program exits, it returns an integer to the operating system called an exit code. Exit codes provide information about whether the program ran successfully or encountered an error. Programs that finish successfully return zero and those that error return a value between 1 and 255. While some of the values in this range have specific meanings, 1 is the catchall for general errors. If you catch an exception in your script that you cannot recover from or if there is some other error and the program must exit, you must set the exit code to a number between 1 and 255 (1 is fine). If your script is interrupted by an unhandled exception python will set the exit code to 1, otherwise, the exit code is zero. However, many other program-specific errors do not cause exceptions, and you be handling all of your exceptions, so you will need to set your exit codes for each error mode. To set the exit code use `sys.exit(1)`.

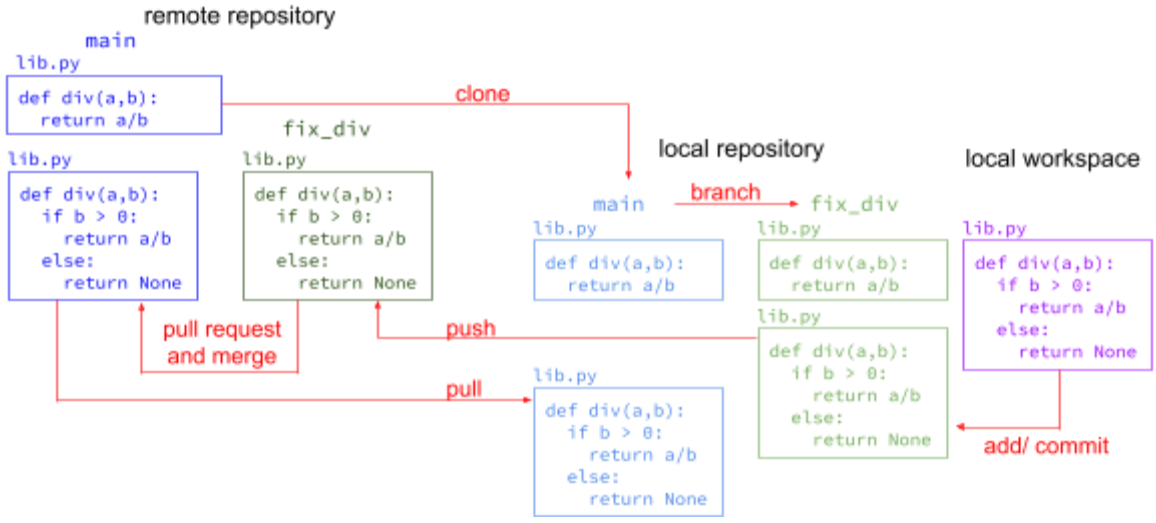
`exit_code.py`

```python
import sys

a = int(sys.argv[1])
b = int(sys.argv[2])

if b == 0:
    sys.exit(1)

print a/b
```

```
$ python exit_code.py  1 0
$ echo  $?
1
$ python exit_code.py  1 1
1
$ echo  $?
0
```

Use the clone branch, commit, push, pull request, merge, tag, pull Git Workflow

To get the most out of version control, never directly commit changes to your local or remote main branch. Instead make all of your changes in branches, then update your remote repo by pushing the branch and creating and merging a pull request. Don't forget to pull those changes to your local main branch.



Use tags to capture a point in our repository's history. Tag names should follow semantic versioning (SemVer) with major, minor, and patch versions (e.g., v1.2.3). Points in your repo that are worth tagging include:

1. **Version milestones**: When you reach certain development milestones that you plan to distribute.
2. **Stable Releases**: Tagging stable versions that are ready for public consumption make it easier for collaborators to find and use the stable version.
3. **Hotfixes**: Tags give easy access to bug fixes in a prior release.
4. **Beta or Alpha Releases**: You can provide early versions of your software to select users for testing. Be sure to tag these versions with appropriate beta or alpha labels (e.g., v1.2.3-beta).

To help remember the **b**ranch, **c**ommit, **p**ush, **p**ull **r**equest, **m**erge, **p**ull workflow, AI has provided use with several mnemonic devices:

Big **C**ats **P**refer **P**laying **M**erry **P**iano

Beautiful **C**ardinals **P**erch, **P**ecking **M**errily, **P**layfully

Brave **C**ows **P**roduce **P**ure **M**ilk **P**rofusely

Big **C**hocolate **P**ies **P**robably **M**ake **P**eople **G**rin

Organize your repository

Your repository is not just a means to storing code. It is the public face of your project. A well-structured repository makes it easier for collaborators to contribute and for users to understand and use your research. The logical organization will also simplify maintenance, even after a long break.

There are many ways to organize your repository, but the basic structure includes the following folders/files:

- **`src` directory** contains all of your projects source code NOT including the tests. For most larger projects this folder will contain sub directories.
- **`test` directory** contains all of your project's test code. Since we have two types of tests, unit tests and functional tests. These different types of tests are distinct enough (written in different languages that test the code at different levels) to warrant future subdivision into two sub directories `unit` and `func`.
- **`doc` directory** contains any additional documentation that you may have for your project. This could include scientific documentation or technical specifications.
- **`.gitignore` file** is optional, but highly recommended to prevent temporary files created by the operating system or your own testing from creeping into your repo. See here for more information https://docs.github.com/en/get-started/getting-started-with-git/ignoring-files
- **`LICENSE` file** is also optional, but without a license, even though the source code might be publicly available, it's not truly "open source." By attaching an open-source license, you are clarifying how others can use your work, under what conditions, and with what obligations. Popular open-source licenses include the MIT License, GNU General Public License (GPL), and the Apache License 2.0, among others. Each has its nuances and implications, so project maintainers should choose one that aligns with their intentions for the project.
- **`README.md` file** is the first thing people see when they encounter your project, and it plays a pivotal role in helping them understand the project's purpose and how to use it. The `.md` (Markdown) format is a plain text file that can be rendered into a more visually appealing format using text-based syntax (more on this next).

Write a good GitHub README

To increase the accessibility and reproducibility of your software, your project must have a good README. READMEs will help users understand what your software does and how to run it. To make writing and updating READMEs easier, and to standardize how READMEs are delivered, GitHub supports `REAMDE.md` files.

A `REAMDE.md` is a markdown file that is automatically rendered by GitHub when a user views a folder in a repository. When a user visits the main page of a repo, the `REAMDE.md` in the root folder is displayed. If the user browses into a sub directory and there is a `REAMDE.md` in that directory, then GitHub will display it. Nothing is displayed for folders without a `REAMDE.md`, including the root.

Markdown syntax https://help.github.com/en/articles/basic-writing-and-formatting-syntax

A markdown file is a text file that can be rendered into a more attractive and easier to read version with the use of a with a few extra formatting symbols. These symbols can control the size, appearance, and organization of the rendered text. For example:

`README.md`

````
# Less is more
## More is less

*somewhat important* or **very important**

Links [Pages](http://github.com/)

- unordered
- list

1. ordered
2. list
  - with
  - nested
    - elements

```
python include_code_snips.sh
```
````

will be rendered as

Less is more

More is less

somewhat important or **very important**

Links Pages

- unorderd
- list

1. ordered
2. list

- with
- nested
    - elements

```
python include_code_snips.sh
```

What to include in your README https://dbader.org/blog/write-a-great-readme-for-your-github-project

At a minimum your README needs to have:

- A short description of what your project does.
    - This is your elevator speech in text form. Keep it brief and mention relevant information which could include the relevant scientific field, statistical models, and input data. Think of the

description as an abstract to the software, which can be more specific than the research abstract in some areas and more general in others.

- How to use your project, with examples.
    - This section can include the usage that is produced by `argparse`, but needs to go into much more detail with specific examples. Go into detail about the format of input files, and give different combinations of input parameters and the resulting software behavior.
- How to install the software.
    - You can never take for granted user system configurations, so give a step-by-step guide that gives specific commands to install all dependencies and run all tests. Conda is useful here because it gives a common starting point.