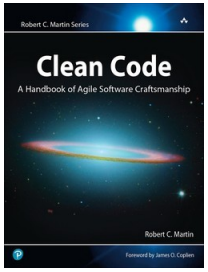
Table of contents (../toc.ipynb)

Clean Code

(https://www.oreilly.com/library/view/clean-code/9780136083238/)



Clean code is a book and a set of coding principles by Robert C. Martin [Martin2008] (../references.bib), aka Uncle Bob (https://blog.cleancoder.com/).

Clean Code aims to foster coding skills of developers for high quality software development.

The coding principles of this book influenced several software developer initiatives like the Software Craftsmanship (https://en.wikipedia.org/wiki/Software_craftsmanship) movement.

The goal of his books is to teach everyone to write beautiful code, which is code that is:

- good and easy to read,
- easy to test,
- easy to reuse,
- easy to modify.

In one sentence, clean code is one method to to reduce technical debt (https://en.wikipedia.org/wiki/Technical_debt). Please consult the book [Kruchten2019] (./references.bib) to learn how to deal with technical debt in software development.

More references

Before we dive into clean code principles, I'd like to point you to some additional references for further study.

Next to clean code book [Martin2008] (../references.bib), there is an entire series of books from Martin to help you to become a better developer: *Clean Architecture* [Martin2018] (../references.bib), *Clean Coder* [Martin2011] (../references.bib)).

A short summary of clean code is to find in a clean code cheat sheet (https://www.planetgeek.ch/wp-content/uploads/2014/11/Clean-Code-V2.4.pdf).

There are many other great books that teach you to write clean code, for instance *The Pragmatic Programmer: From Journeyman to Master* [Hunt2019] (../references.bib), or *Clean Code in Python - Refactor Your Legacy Code Base* [Anay2018] (../references.bib).

Clean Code principles

First, let us have a look at three quotes to introduce clean code.

Quote one

"*Writing clean code is what you must do in order to call yourself a professional. There is no reasonable excuse for doing anything less than your best*" [Martin2008] (../references.bib).

Quote two

"*Programming is the art of telling another human being what one wants the computer to do.*"
[Donald Knuth]

Quote three

"*Computers are good at following instructions, but not at reading your mind.*" [Donald Knuth]

Almost anything what follows can be derived from the quotes of Donald Knuth. Just keep in mind that you write code for another human, not for the computer and humans spend most time for reading code not for writing code in larger software projects.

Clean Code in a nutshell

Now that we know that code is written for humans, the following list of main principles in clean code should make sense. This list is not comprehensive but contains my favorites.

Use meaningful names.

- This is true for all layers of code. Please use meaningful names for folders, classes, functions, variables, tests...
- Meaningful names are for instance very handy if you start to search for something within a large project.

Functions should:

- **be small**,
- **do one thing**,
- use descriptive names,
- have no side effects,
- have not more than three arguments (best is zero).

Methods in class can be treated similarly to functions. And **classes** should be small as well.

Do not repeat yourself!

- Avoid redundant code.
- Reuse code instead.

If you start to repeat yourself, you will end up with something which is known as Spaghetti code (https://en.wikipedia.org/wiki/Spaghetti_code).

Comments do not heal bad code.

Comments are often outdated compared with the code itself. Instead of using comments, try to express yourself in the code! Fewer is better here.

Good comments are

- informative,
- explain the intention why something has been made,
- warn of consequences,
- list ToDos.

Bad comments are

- redundant,
- misleading, outdated, or simply wrong,
- comments that comment out code.

Use good readable formatting!

Well you are in a Python course, so if you follow PEP 8 -- Style Guide for Python Code [PEP8] (../references.bib) your code will be superior readable from a formatting point of view.

Formatting means vertical and horizontal space. In spite of other languages, Python was designed with formatting in mind. So you should be fine with this principle by default.

”Why clean code, we have Continuous Integration!”

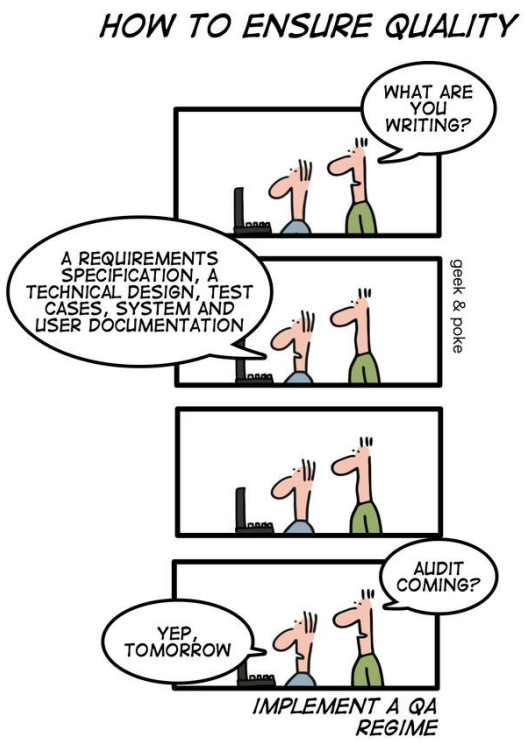
Yes you can and should automate whenever possible and hence you can automate static code analysis tests, and linting. But clean code is more than linted code.

Clean code is good readable and maintainable. Note that roughly 80% of time is required for developers to read code. Hence, a linter can tell you that a variable should have lower case letters only, but it can not suggest a meaningful variable name.

Therefore, **if you automate a mess, you get automated mess**.

(http://geek-and-poke.com/geekandpoke/2010/9/7/how-to-ensure-quality.html)

Beyond Clean Code



Clean Code contributes to software quality but clean code is not enough to ensure high software quality.

On a higher level, you need **clean architecture, clean tests, clean design, and even clean organization** as well to develop a product on high quality.

An example of a not so well defined organization is if subtasks of a product like implementation, testing, and marketing are dispersed over different business units which do not communicate with each other.

Think about the mess which comes up if the product manage from unit A sells a feature which is not yet developed in unit B and which will be forgotten to test in unit C. This **organizational mess will cause a messy product**.

Clean Code example

We will take a look at a typical Python script of an engineering task and start with a version, which violates as much as possible clean code principles.

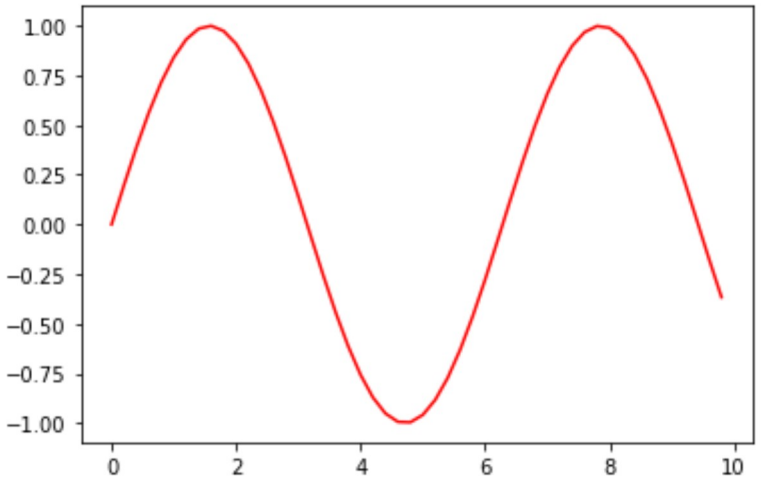
The next steps are small code improvements to transform the code into clean code and this work is known as **refactoring** in software engineering. Refactoring (https://en.wikipedia.org/wiki/Code_refactoring) is nothing else than improvements of the code without any changed functionality. Therefore, refactoring improves the code smell and reduces technical debt.

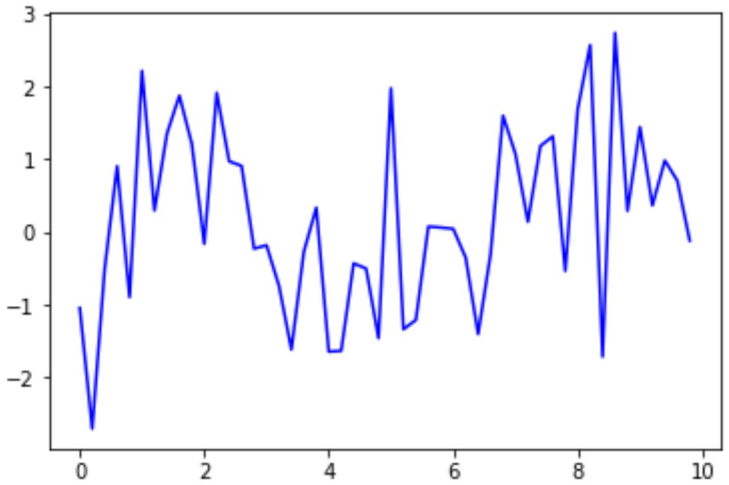
The fact that refactoring does not change or add functionality **can be a source of conflicts** if quantity of delivered features is more important than quality. If you end up in a project that does not preserve development time for refactoring, technical debt will increase and quality of software will rot.

Now let us take a look at the example script and try to read it.

In [23]:
```
# %load bad_smelling_code.py
import numpy as np;import matplotlib.pyplot as plt
x=np.arange(0,10,0.2);y=np.sin(x)
plt.figure();plt.plot(x,y,'r')
plt.figure();plt.plot(x,y+np.random.randn(len(x)),'b')
z=np.sin(x)+3.0
plt.figure();plt.plot(x,z,'r')
plt.figure();plt.plot(x,z+np.random.randn(len(x)),'b')
```

Out[23]:
```
[<matplotlib.lines.Line2D at 0x7f9bf4599350>]
```





I did my best to write the poorest Python script ever. Let us try to read the code, because we know now that code is actually written for humans not for the computer.

```
import numpy as np;import matplotlib.pyplot as plt
x=np.arange(0,10,0.2);y=np.sin(x)
plt.figure();plt.plot(x,y,'r')
plt.figure();plt.plot(x,y+np.random.randn(len(x)),'b')
z=np.sin(x)+3.0
plt.figure();plt.plot(x,z,'r')
plt.figure();plt.plot(x,z+np.random.randn(len(x)),'b')
```

First, it is very hard to read because indentation and whitespace is missing. But if you take your time you will see that the script does not that much. It just generates data, does a plot, does another plot, generates nearly the same data, does a similar plot, and finally, plot this data with noise once more.

I hope you get a bit tired form reading this script and also get bored, because stuff repeats and copy and paste was used a lot.

Refactoring

We will change now the coding style to end up with a better code smell without changing the functionality of the code.

Formatting

One Clean Code principle was to use good formatting. Let us add horizontal and vertical white space to improve readability and give this script more structure.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 10, 0.2)
y = np.sin(x)

plt.figure()
plt.plot(x, y, 'r')

plt.figure()
plt.plot(x, y + np.random.randn(len(x)), 'b')

z = np.sin(x) + 3.0

plt.figure()
plt.plot(x, z, 'r')

plt.figure()
plt.plot(x, z + np.random.randn(len(x)), 'b')
```

The additional horizontal white space makes the code easier to read.

Although there are now more lines of code, the additional vertical white space helps to discover redundant code. All the plot commands are very similar and can be written as function.

Do not repeat yourself!

We will use now a function to program all the plots and to avoid copy and paste of code as much as possible.

```
import matplotlib.pyplot as plt
import numpy as np


def my_plot(x, y):
    plt.figure()
    plt.plot(x, y, 'r')
    plt.figure()
    plt.plot(x, y + np.random.randn(len(x)), 'b')


x = np.arange(0, 10, 0.2)
y = np.sin(x)
z = np.sin(x) + 3.0

my_plot(x, y)
my_plot(x, z)
```

Now the script has tree sections: a plot part, some code which generates data, and the section where the plot is called twice.

Use meaningful names

Please have a look at this version, where the variable names are more descriptive.

```
import matplotlib.pyplot as plt
import numpy as np


def my_plot(time, values, noise):
    plt.figure()
    plt.plot(time, values, 'r')
    plt.figure()
    plt.plot(time, values + noise, 'b')


TIME = np.arange(0, 10, 0.2)
VAL = np.sin(TIME)
VAL2 = np.sin(TIME) + 3.0

my_plot(time=TIME, values=VAL, noise=np.random.randn(len(TIME)))
my_plot(time=TIME, values=VAL2, noise=np.random.randn(len(TIME)))
```

Now, even without a single comment or description in the code it is obvious that the `my_plot` function takes now three arguments, which are called `time`, `values`, `noise`. The difference in the two times called plot is in the passed values.

We can even improve the noise generation with a small function.

```
import matplotlib.pyplot as plt
import numpy as np


def my_plot(time, values):
    plt.figure()
    plt.plot(time, values, 'r')
    plt.figure()
    plt.plot(time, add_noise_to(values), 'b')


def add_noise_to(values):
    return values + np.random.randn(len(values))


TIME = np.arange(0, 10, 0.2)
VAL = np.sin(TIME)
VAL2 = np.sin(TIME) + 3.0

my_plot(time=TIME, values=VAL)
my_plot(time=TIME, values=VAL2)
```

And finally, the lines of code which create the sine functions can also be written as small functions. This makes the plot function more general because other functions could be passed now as well.

```
import matplotlib.pyplot as plt
import numpy as np


def my_plot(time, func):
    plt.figure()
    plt.plot(time, func(time), 'r')
    plt.figure()
    plt.plot(time, add_noise_to(func(time)), 'b')


def sine_wave(time):
    return np.sin(time)


def biased_sine_wave(time):
    return sine_wave(time) + 3.0


def add_noise_to(values):
    return values + np.random.randn(len(values))


TIME = np.arange(0, 10, 0.2)

my_plot(time=TIME, func=sine_wave)
my_plot(time=TIME, func=biased_sine_wave)
```

This last version with simple functions is easy to read, easy to test, and simple to extend. Although the number of code lines grew, I would prefer this solution instead of the bad_smelling_code (./bad_smelling_code.py) file.

Tools to rate code smell

You can use flake8 (https://pypi.org/project/flake8/), pylint (https://www.pylint.org/), and of course inbuilt features of IDEs like Pycharm to rate the code smell automatically. These static code analysis cannot detect all bad programming parts but most of common mistakes.

We will use `flake8` and `pylint` in a small exercise to see what this static code analysis returns.

Exercise: `flake8` and `pylint` (3 minutes)

- Download the file bad_smelling_code (./bad_smelling_code.py).
- Run in terminal `pylint bad_smelling_code.py`.
- Also use flake8 with `flake8 bad_smelling_code.py`.

Result

My run of `pylint` returned a lot of problems and rated the code at

---

> *Your code has been rated at -7.69/10*

Minus 7.69 is just horrible!

So, you can save a lot of time (reading terrible code) if you apply static code analysis in your project.