Clean code

for Python beginners

What is clean code?

Why clean code is necessary*?*

How to put Clean Code into practice?

Where to begin?

What is clean code

Clean code, code quality, code style are interrelated terms.

Clean code is the concept that sets the bar on how good code should look and operate.

It includes values, principles and rules that are necessary to follow in order to produce high quality code.

Following the code style of a language is only a part of clean code.

Writing code is easy,
but writing good, clean code is hard.

Code is considered to be of high quality when:

- It serves its purpose
- It's behavior can be tested
- It follows a consistent style
- It's understandable
- It doesn't contain security vulnerabilities
- It's documented well
- It's easy to maintain

Why clean code is valuable

From the practical side, Clean code is a skill, helping to produce high-quality code, which includes keeping our code readable, maintainable, and extendable.

Clean code is an important aspect of writing quality software.

Knowing how to write clean code is a great skill to add to your CV and demonstrate to your colleagues and employers.

Clean code is necessary

Programming is not just about writing code, but also about reading and working with code written by others.

As a software developer you're very likely working in a team. And, in a team setting, it's very important that all developers follow the same coding standards. Otherwise, it's much harder to read someone else's code.

Clean code style is not a formality: it actually helps avoid a lot of problems.

Readability counts. - The Zen of Python

Code quality

Code quality generally refers to how functional and maintainable your code is.

How do we measure it?

We can define the level of code quality, depending on the number of detected issues, their priorities, and the size of the whole program..

There can be distinguished 4 groups of issues that affect code quality:

- ***Code style issue*** means that your code violates one of the rules recorded in the style guide for the language you're using.
- ***Best practice issue*** means that your code does not follow the widely accepted recommendations. Some features of the language can be used in an inefficient or obsolete way. Such errors are not critical but the fewer of them, the better.
- ***Error-prone issue*** **means** that your code contains a potential bug.

Error-prone issues should always be fixed regardless of the size of your program. Even if your code passes unit tests, it may behave incorrectly in some cases, which would be a problem in future.

- ***Code complexity issue*** means that your solution is poorly designed or overly complicated. Reducing code complexity makes it easier to understand, edit, and debug any given part of your program. High code complexity means that some part of your program contains too many conditions, loops, logical operators; in other words, it does too many things.

Where to begin

Tip №1: make use of certain tools.

Checking for errors affecting code quality can be handled by a computer via special software - linters, code formatters, and security vulnerability scanners.

Linters

A code **linter** is basically a program that inspects your code and gives feedback. A linter can tell you the issues in your program and also, a way to resolve them. You can run it anytime to ensure that your code is matching standard quality.

The linters are configured in accordance with the style guides of the corresponding languages .

Linters flag programming errors, bugs, stylistic errors, and suspicious constructs through source code analysis.

Linters look at aspects of code and detect lints:

- **Logical Lint:** tells about code errors, dangerous code patterns
- **Stilystical Lint:** looks at formatting issues

Linters flag programming errors, bugs,]

The most popular Python linters are:

- pylint
- flake8 (includes complexity checking)

These linters can be integrated into various IDEs, they are highly configurable, customizable and you can easily adds many features.

For example, to enforce PEP-8 naming conventions, install pep8-naming:

$ pip install pep8-naming

Tip №2: pay attention to style conventions

Quote:

"I remember to have had learned pep8 (the python standard) by heart like poetry.

I even had it hung over my fridge in the university dorm. The standard was of course, just the beginning."

Python Code Style conventions

- Do not use more than 79 characters in a line of code.
- Avoid extra spaces within parentheses ( ).
- Avoid an extra space before an open parenthesis (.

Python Code Style conventions

- use either single ' or double " quotes to define strings. Choose one (' or ") and consistently use it in your code. if a string contains single or double quotes, you should use the other one to avoid backslashes.

Bad	**Good**
`print('It\'s a bad string!')` - harder to read. Backslash in this case is an escape character;	`print("It's a good string!")`

Python Code Style conventions

Naming rules for variables:

1. Use lowercase and underscores _ to split words.

Bad	**Good**
`httpresponse` `myVariable`	`http_response`

2. If the most suitable variable name is some Python keyword, add an underscore to the end of it.

Bad	**Good**
`klass = type(var)`	`class_ = type(var)`

Python Code Style conventions

Naming rules for variables:

3. Although you can use any Unicode symbols, limit variable names with ASCII characters.

Bad	**Good**
`copy = "And I'm written in Cyrillic!"`	`copy = "I'm written in Latin alphabet”`

4. Avoid names from the built-in types list.

`str = 'Hello!'` is a bad idea,

because in the further code you can't use str type as it's overridden

Python Code Style conventions

Conventions for docstrings:

1. """ triple-double """ quotes are punctuation signs to indicate a docstring. The annotation should start with a Capital Letter and end with a period . (as recommended by PEP 257).
2. each line in a docstring should be no longer than 72 characters (just like a comment).
3. """

   The opening and the closing quotes should be on the same line.

   """

Python Code Style conventions

Conventions for docstrings:

4. The docstring may start right after the triple-double quotes """. But it is also possible to specify them on the next line after the opening quotes
5. The detailed description starts at the same position as the first quote """ of the first docstring line — there's no indent.
6. no empty strings either before or after the """Docstring"""

Python Code Style conventions

Conventions for docstrings:

7. summary in the """Docstrings""" is separated from detailed description by a single blank line.
8. description in the """Docstrings"""should be imperative: we need the wordings like """Return the factorial.""" instead of """Returns the number.""".
9. The description in a """Docstring""" is not a scheme that repeats the object's parameters and returned values:

Python Code Style conventions

Conventions for docstrings :

Bad:

```
def count_factorial(num):
        """count_factorial(num) -> int."""“
```

Good:

```
def count_factorial(num):
        """Return the factorial of the number.

        Arguments:
        num -- an integer to count the factorial of.
        Return values:
        The integer factorial of the number.
        """
        if num == 0:
            return 1
        else:
            return num * count_factorial(num-1)
```

Python Code Style conventions

Conventions for docstrings:

For class and module docstrings PEP 257 proposes the following conventions:

- Module docstrings should also provide a brief one-line description. After that, it is recommended to specify all classes, methods, functions, or any other of the module's objects.
- insert a blank line after a class docstring to separate the class documentation and the first method

Python Code Style conventions

Example of module docstring:

```
# information.py module
"""The functionality for manipulating the user-related information."""

class Person:
    """The creation of the Person object and the related functionality.
    """

    def __init__(self, surname, birthdate):
        """The initializer for the class.

        Arguments:
        surname -- a string representing the person's surname.
        birthdate -- a string representing the person's birthdate.
        """
        self.surname = surname
        self.birthdate = birthdate

    def calculate_age(self):
        """Return the current age of the person."""
        # the body of the method
```

Tip №2: Consistency. Set a few rigid rules for yourself and follow them.

Quote: *It's better to begin from just a few rules:*

- Do not allow any method to exceed 15 lines. Ever.
- Never repeat 5 lines of code. If 5 lines of code repeat, put them in their own method.
- Never write a class, method, or variable name that does not consist of one or more full words that describe what it is for. If your loop variable is named i, change it to index. Don't let it paralyze you: give it a bad name and change it later, but make it descriptive.

Tip №3: algorithm before coding

Quote:

- *Don't read the problem statement or requirements and immediately start punching code. Spend quality time designing, correcting, analyzing and beautifying your algorithm before coding*
- *Create Pseudo-code - write out your algorithm(s) before you get in front of an IDE*

Tip №4: learn some design principles

Quote:

"Familiarize yourself with SOLID principles. Don't panic - I didn't say become an expert, just find out what they are. Then set the last 4 aside and focus on the S - Single Responsibility Principle, because it's the easiest to understand and provides immediate benefit."

Principles of object-oriented programming and design are intended to help developers to write maintainable extensible code. Adopting these practices can also contribute to avoiding code smells, refactoring code, and Agile or Adaptive software development.

OOP design principles

Single Responsibility Principle('S' in SOLID principles): a module, a class or a function has to only do 1 thing. In other words, they have to have only one responsibility.

Why? This way the code is more robust, easier to debug, read and reuse.

The error of many novice programmers is to write complex functions and classes that do a lot of things.

The function, shown below, takes a word and a file path as parameters and returns a ratio of number of the word's occurrences in the text to the total number of words:

OOP design principles

```
def percentage_of_word(search, file):
    search = search.lower()
    content = open(file, "r").read()
    words = content.split()
    number_of_words = len(words)
    occurrences = 0
    for word in words:
        if word.lower() == search:
            occurrences += 1
    return occurrences/number_of_words
```

- This code does many things in 1 function: reads file, calculates number of total words, number of word's occurrences, and then returns the ratio.

If we want to follow the Single Responsibility Principle, we need to refactor our code like that:

OOP design principles

```
def read_localfile(file):
    '''Read file'''

    return open(file, "r").read()


def number_of_words(content):
    '''Count number of words in a file'''

    return len(content.split())


def count_word_occurrences(word, content):
    '''Count number of word occurrences in a file'''

    counter = 0
    for e in content.split():
        if word.lower() == e.lower():
            counter += 1
    return counter
```

OOP design principles

```
def percentage_of_word(word, content):
    '''Calculate ratio of number of word occurrences to number of all
    words in a text'''

    total_words = number_of_words(content)
    word_occurrences = count_word_occurrences(word, content)
    return word_occurrences/total_words


def percentage_of_word_in_localfile(word, file):
    '''Calculate ratio of number of word occurrences to number
    of all words in a text file'''

    content = read_localfile(file)
    return percentage_of_word(word, content)
```

How to apply Clean Code principles

How to put Clean Code principles into practice?

When we write something, we should review our code on a regular basis, clean it up and try to improve it.

Even beginners in programming can put some simple Clean Code principles into practice right from the start:

- **Use comments for clarification.** It is better to use 1 or 2 lines for a comment with explanation than to force people to guess why we implement this or that function or method or why we created it in that specific way. Meaning, the history may be still unclear.

How to apply Clean Code principles

- **Don't repeat yourself** (DRY) principle basically means: Do your absolute best to avoid duplicate code.

Duplicate code is bad because it means that there's more than 1 place to alter something if you need to change some logic. Often you have duplicate code because you have 2 or more slightly different things, that share a lot in common, but their differences force you to have 2 or more separate.

When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements.

How to apply Clean Code principles

- **Do not over- minify our code**: it is not necessary to write code that looks like minified. Instead, we can use indentation, line breaks and empty lines to make the structure of our code more readable.
- **spaces are preferred over tab (**according to PEP 8). '*For new projects, spaces-only are strongly recommended over tabs.*' - This is a clear recommendation, and a strong one, but not a prohibition of tabs. Note that the use of tabs confuses another aspect of PEP 8: Limit all lines to a maximum of 79 characters. Tabs should be used solely to remain consistent with code that is already indented with tabs. Python disallows mixing tabs and spaces for indentation.

Clean Code concept in examples

- *from Robert C. Martin's book adapted for Python*

This is not a style guide.

It's a guide to producing readable, reusable, and refactorable software in Python.

Now let's go into some detailes with bad/good code examples for:

- *variables*
- *functions*

Variables

Use meaningful and pronounceable variable names

Bad:

```
import datetime


ymdstr =
datetime.date.today().strftime("%y-%m-
%d")
```

Good:

```
import datetime


current_date: str =
datetime.date.today().strftime("%y-%m-%d")
```

Use the same vocabulary for the same type of variable

Bad: Here we use three different names for the same underlying entity

```
def get_user_info(): pass
def get_client_data(): pass
def get_customer_record(): pass
```

Good: If the entity is the same, you should be consistent in referring to it in your functions:

```
def get_user_info(): pass
def get_user_data(): pass
def get_user_record(): pass
```

Variables

Use searchable names

We will read more code than we will ever write.

Bad:

```
import time


# What is the number 86400 for again?
time.sleep(86400)
```

Good:

```
import time


# Declare them in the global namespace for
the module.
SECONDS_IN_A_DAY = 60 * 60 * 24
time.sleep(SECONDS_IN_A_DAY)
```

Variables

Use explanatory variables

Bad:

```
import re


address = "One Infinite Loop, Cupertino
95014"
city_zip_code_regex =
r"^[^,\\]+[,\\\s]+(.+?)\s*(\d{5})?$"

matches = re.match(city_zip_code_regex,
address)
if matches:
    print(f"{matches[1]}: {matches[2]}")
```

Not ***bad***: It's better, but we are still heavily dependent on regex.

```
import re


address = "One Infinite Loop, Cupertino
95014"
city_zip_code_regex =
r"^[^,\\]+[,\\\s]+(.+?)\s*(\d{5})?$"
matches = re.match(city_zip_code_regex,
address)

if matches:
    city, zip_code = matches.groups()
    print(f"{city}: {zip_code}")
```

Variables

Use explanatory variables

Good: Decrease dependence on regex by naming subpatterns.

```
import re


address = "One Infinite Loop, Cupertino 95014"
city_zip_code_regex = r"^[^,\\]+[,\\\s]+(?P<city>.+?)\s*(?P<zip_code>\d{5})?$"

matches = re.match(city_zip_code_regex, address)
if matches:
    print(f"{matches['city']}, {matches['zip_code']}")
```

Variables

Avoid Mental Mapping

Don't force the reader of your code to translate what the variable means. Explicit is better than implicit.

Bad:

```
seq = ("Austin", "New York", "San
Francisco")

for item in seq:
    #do_stuff()
    #do_some_other_stuff()

    # Wait, what's `item` again?
    print(item)
```

Good:

```
locations = ("Austin", "New York", "San
Francisco")

for location in locations:
    #do_stuff()
    #do_some_other_stuff()
    # ...
    print(location)
```

Don't add unneeded context

If your class/object name tells you something, don't repeat that in your variable name.

Bad:

```
class Car:
    car_make: str
    car_model: str
    car_color: str
```

Good:

```
class Car:
    make: str
    model: str
    color: str
```

Functions

Stop writing Python functions that take more than 3 minutes to understand

Function arguments (2 or fewer)

Limiting the amount of function parameters makes testing your function easier. Having more than 3 leads to a combinatorial explosion where you have to test tons of different cases with each separate argument.

Usually, if you have more than 2 arguments then your function is trying to do too much. In cases where it's not, most of the time a higher-level object will suffice as an argument.

Bad:

```
def create_menu(title, body,
button_text, cancellable):
    pass
```

Java-**esque**:

```
class Menu:
    def __init__(self, config: dict):
        self.title = config["title"]
        self.body = config["body"]
        # ...

menu = Menu(
    {
        "title": "My Menu",
        "body": "Something about my menu",
        "button_text": "OK",
        "cancellable": False
    }
)
```

Every function should do 1 thing and do it well

When you can isolate a function to just 1 action, they can be refactored easily and your code will read much cleaner. If you take nothing else away from this guide other than this, you'll be ahead of many developers.

Bad:

```
from typing import List

class Client:
    active: bool

def email(client: Client) -> None:
    pass

def email_clients(clients: List[Client])
-> None:
    """Filter active clients and send
    them an email.
    """
    for client in clients:
        if client.active:
            email(client)
```

Good:

```
from typing import List

class Client:
    active: bool

def email(client: Client) -> None:
    pass

def get_active_clients(clients: List[Client]) ->
List[Client]:
    """Filter active clients.
    """
    return [client for client in clients if
    client.active]

def email_clients(clients: List[Client]) ->
None:
    """Send an email to a given list of
    clients.
    """
    for client in get_active_clients(clients):
        email(client)
```

Do you see an opportunity for using generators now?

Functions

Every function should do 1 thing and do it well

Even better

```
from typing import Generator, Iterator


class Client:
    active: bool


def email(client: Client):
    pass

def active_clients(clients: Iterator[Client]) -> Generator[Client, None, None]:
    """Only active clients"""
    return (client for client in clients if client.active)

def email_client(clients: Iterator[Client]) -> None:
    """Send an email to a given list of clients.
    """
    for client in active_clients(clients):
        email(client)
```

Function names should say what they do

Bad:

```
class Email:
    def handle(self) -> None:
        pass

message = Email()
message.handle() # What is this supposed
to do again?
```

Good:

```
class Email:
    def send(self) -> None:
        """Send this message"""

message = Email()
message.send()
```

Don't use flags as function parameters

Flags tell your user that this function does more than one thing. Functions should do one thing. Split your functions if they are following different code paths based on a boolean.

Bad:

```
from typing import Text
from tempfile import gettempdir
from pathlib import Path


def create_file(name: Text, temp: bool)
-> None:
    if temp:
        (Path(gettempdir()) /
        name).touch()
    else:
        Path(name).touch()
```

Good:

```
from typing import Text
from tempfile import gettempdir
from pathlib import Path


def create_file(name: Text) -> None:
    Path(name).touch()


def create_temp_file(name: Text) -> None:
    (Path(gettempdir()) / name).touch()
```

Functions

Avoid side effects

A function produces a side effect if it does anything other than take a value in and return another value or values. For example, a side effect could be writing to a file, modifying some global variable, using an instance of a class, and not centralizing where your side effects occur.

Bad:

```
# type: ignore
# This is a module-level name.
# It's good practice to define these as immutable values, such as a string.
# However...

fullname = "Ryan McDermott"

def split_into_first_and_last_name() -> None:
# The use of the global keyword here is changing the meaning of the
the following line. This function is now mutating the module-level
state and introducing a side-effect!
    global fullname
    fullname = fullname.split()
```

Functions

```
split_into_first_and_last_name()
print(fullname)  # ["Ryan", "McDermott"]

# MyPy will spot the problem, complaining about 'Incompatible types in
assignment: (expression has type "List[str]", variable has type "str")'
# OK. It worked the first time, but what will happen if we call the function
again?
```

Good:

```
from typing import List, AnyStr


def split_into_first_and_last_name(name: AnyStr) -> List[AnyStr]:
    return name.split()

fullname = "Ryan McDermott"
name, surname = split_into_first_and_last_name(fullname)

print(name, surname)  # => Ryan McDermott
```

Clean Code concept in examples

to sum up, Clean Code suggests that good variables have:

- meaningful and pronounceable names
- the same vocabulary for the same underlying entity
- searchable names, no unneeded context

Clean Code suggests your Python function to:

- do 1 thing, be small.
- have fewer than 4 arguments.
- not use flags as function parameters
- have no duplication
- use descriptive names
- avoid side effects
- contain code with the same level of abstraction

about the Best Practices

Quote:

“We have a document with all of the Best Practices listed by category. “

“Pythonic” way of writing code means that yoy’re writing Python in the way that the language was intended to be written. And there’s a lot to cover here.

Best practices are not supposed to be accessed and learned at once, but developers have to learn and follow them in order to write clean and professional code.

Conclusion

Code quality is one of the most opinionated topics in software development.

Code style, in particular, is a sensitive issue amongst developers since we spend much of our development time reading code. It's much easier to read and infer intent when code has a consistent style that adheres to PEP-8 standards.

Good news is that detecting errors and correcting code style issues can be handled by a computer via special tools. But that’s not enough to ensure high quality of your code.

Another great thing to look at is Clean code concept: applying its principles is the key to producing high quality code.

Many thanks and credits to:

Nik Tomazic https://testdriven.io/blog/clean-code-python/

Jan Giacomelli https://testdriven.io/blog/python-code-quality/

Anjali Jaiswal https://anjali-jaiswal.medium.com/solid-principles-for-beginners-857abd8ffb3d

Gleb Tocarenco https://www.pentalog.com/blog/it-development-technology/clean-code-with-python

ALEX DEVERO https://blog.alexdevero.com/6-simple-tips-writing-clean-code/

Khuyen Tran https://towardsdatascience.com/python-clean-code-6-best-practices-to-make-your-python-functions-more-readable-7ea4c6171d60

https://www.quora.com/ members

https://github.com/zedr/clean-code-python

This presentation is prepared by Anastasia Klimenko

https://github.com/gelst13