Clean code

Rough agenda:

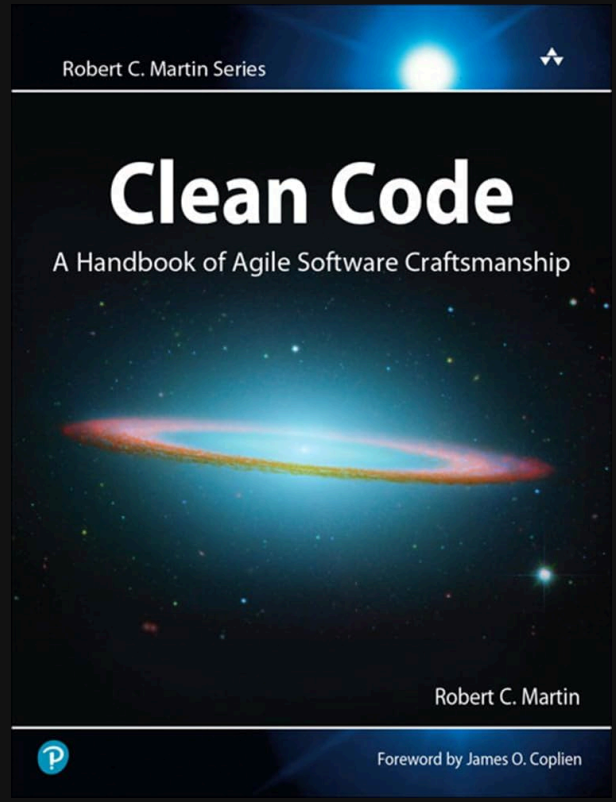
1. Definition
2. General advice
3. Formatting & Linting

Definition

> Code is clean if it can be understood easily – by everyone on the team.
>
> Clean code can be read and enhanced by a developer other than its original author. With understandability comes readability, changeability, extensibility and maintainability.

The term was coined by Robert C. Martin's book. It contains some really solid advice but also some polarizing opinions.

Don't be dogmatic about anything when it comes to this topic! It's very easy to fall for this trap and get hung up in pointless arguments.



General advice

1. Follow standard conventions.
2. KISS: Keep it simple stupid. Simpler is always better. Reduce complexity as much as possible.
3. Scout rule: Leave the campground cleaner than you found it.
4. Always look for the root cause of a problem.

General advice

Follow standard conventions

For Python, a widely used convention is the language's official styleguide PEP 8.

It's not necessary that you strictly follow exactly this exact styleguide!

Especially for smaller projects you might not even need a styleguide at all. However, if we talk about any larger production codebase that actively maintained by many developers: It's really handy to have one.

How to ensure that everyone follows the styleguide?

Option 1: During code review.
=> Annoying for both the reviewer and the dev. Both end up hating each other.

Option 2: Automate it using: Linting and Formatting!
=> This will catch most convention-breaking code. The rest is up to the reviewer.

Linting and formatting

How?

For Python, we recommend to take a look at ruff. It's a useful all-in-one tool. Not so long ago, you had to use a bunch of different tools (Flake8, Black, isort and more) for this!

Linter output:

```
uv run ruff check
```

```
> main.py:1:16: F401 [*] `os.path` imported but unused
>   |
> 1 | from os import path
>   |                ^^^^ F401
>   |
>   = help: Remove unused import: `os.path`
>
> Found 1 error.
> [*] 1 fixable with the `--fix` option.
```

Formatter in action:

```
from os import path
num_terms    =10
result= calculate_fibonacci(  num_terms)
for i,         num in         enumerate(result):
  print(f"Term {i+1}: {num     }")

def calculate_fibonacci(n):
    if n <= 0:
                return []
    elif n == 1:
        return [0]
    sequence = [0, 1]
    while len(sequence) < n:
                    sequence.append(sequence[-1] + sequence[-2])
    return      sequence
```

Linting and formatting

How?

For Python, we recommend to take a look at ruff. It's a useful all-in-one tool. Not so long ago, you had to use a bunch of different tools (Flake8, Black, isort and more) for this!

Linter output:

```
uv run ruff check
```

```
> main.py:1:16: F401 [*] `os.path` imported but unused
>   |
> 1 | from os import path
>   |                ^^^^ F401
>   |
>   = help: Remove unused import: `os.path`
>
> Found 1 error.
> [*] 1 fixable with the `--fix` option.
```

Formatter in action:

```python
from os import path

num_terms = 10
result = calculate_fibonacci(num_terms)
for i, num in enumerate(result):
    print(f"Term {i + 1}: {num}")

def calculate_fibonacci(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    sequence = [0, 1]
    while len(sequence) < n:
        sequence.append(sequence[-1] + sequence[-2])
    return sequence
```

Linting and formatting

When?

Formatting

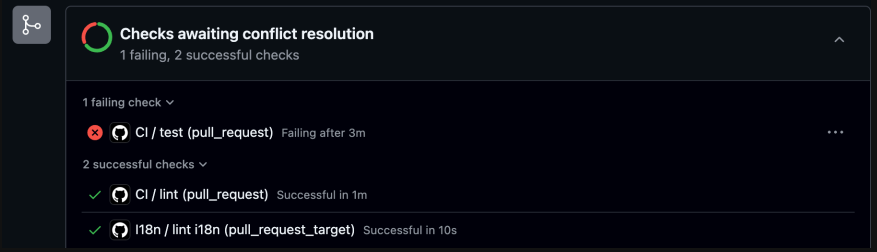
A popular choice is: format on file-save. If you're using VSCode, you can easily use an extension for this.

Linting

This is more tricky. We can easily run `ruff check` but how do we ensure to not forget it from time to time?

You can set up a githook that checks for linter-errors before every `git commit` or `git push`.
Now you can be sure that **you** don't forget about linting.

But what about other devs? You can't force them to set up githooks on their local machine.
This is where GitHub Actions shine. You can use them to check for linter-errors upon pull requests. This is part of continuous integration (CI).



General advice

KISS: Keep it simple stupid

> = Don't unnecessarily complicate things if there is an easier way to do it.

The definition of *complicated* may vary within and across teams. Maybe you are a Python wizard/witch but:

1. is everyone else who's currently in your team?
2. will the future maintainer of the your code understand it?
3. do you really need 3 nested dict-comprehensions ?!

More often than not, it's actually harder to write *simple* code rather than *complicated* code. It forces you have an in-depth understanding of what your are doing. There are whole programming languages based on this principle (e.g. Go).

> Simplicity carried to an extreme becomes elegance.
>
> -- Jon Franklin (i think)

General advice

Scout rule: Leave the campground cleaner than you found it.

If you contribute to a codebase and you stumble accross some legacy or buggy code: Make an effort to refactor it!

It does not matter if you wrote the old code or somebody else.

General advice

Always look for the root cause of a problem.

This mainly applies to bug fixing. Trying to fix a bug only by fixing it's symptoms, will only lead to the bug popping up again a few days later.

You can also translate this advice into other areas. E.g. product management:
If your users are not using your shiny new feature, find out why and don't force them to use it!

Most comments are overrated

Use self-documenting code instead

A very basic example for this is proper function and variable names. Just be descriptive in your naming:

```
def calc(n, t):
    # Calculates total price after tax
    return n * (1 + t/100)
```

> Content recommendation: This video goes into more detail about exactly this topic.

Most comments are overrated

Use self-documenting code instead

A very basic example for this is proper function and variable names. Just be descriptive in your naming:

```
def calculate_price_with_tax(base_price, tax_rate):
    return base_price * (1 + tax_rate/100)
```

> Content recommendation: This video goes into more detail about exactly this topic.

Most comments are overrated

Proper typing is sometimes enough!

```python
def merge_user_data(users, profiles):
    """
    Combines user accounts with their profile information.
    users: list of dicts with 'id' and 'email'
    profiles: list of dicts with 'user_id' and 'bio'
    returns: list of complete user records
    """
    lookup = {p['user_id']: p['bio'] for p in profiles}
    return [{**user, 'bio': lookup.get(user['id'], '')} for user in users]
```

For more advanced use cases, Pydantic models are really useful. More on that later!

Most comments are overrated

Proper typing is sometimes enough!

```python
from typing import List, Dict, TypedDict

class PartialUser(TypedDict):
    id: int
    email: str

class Profile(TypedDict):
    user_id: int
    bio: str

class User(TypedDict):
    id: int
    email: str
    bio: str

def merge_user_data(users: List[PartialUser], profiles: List[Profile]) -> List[User]:
    lookup = {p['user_id']: p['bio'] for p in profiles}
    return [{**user, 'bio': lookup.get(user['id'], '')} for user in users]
```

For more advanced use cases, Pydantic models are really useful. More on that later!

Most comments are overrated

but sometimes you really need them

Sometimes, there really is no way to improve your code (especially given time constraints) but you feel like some important information is missing.
=> Commenting is good in this case

However, most of the time if you can’t rewrite or enhance your code to make comments redundant:
Your whole problem-solving approach is probably flawed and you should look for an entirely different way to code the solution.

DRY is overrated

The DRY principle stands for: Don't repeat yourself

Of course, you should make use of it for obvious cases:

```
def welcome_john():
    print("Hello John!")
    print("Welcome to our app!")
    print("Enjoy your stay!")

def welcome_mary():
    print("Hello Mary!")
    print("Welcome to our app!")
    print("Enjoy your stay!")
```

DRY is overrated

The DRY principle stands for: Don't repeat yourself

Of course, you should make use of it for obvious cases:

```
def welcome(name):
    print(f"Hello {name}!")
    print("Welcome to our app!")
    print("Enjoy your stay!")
```

DRY is overrated

"Duplication is far cheaper than the wrong abstraction" – Sandi Metz.

1. Programmer A sees duplication.

2. Programmer A extracts duplication and gives it a name.
   => This creates a new abstraction. It could be a new method, or perhaps even a new class.

3. Programmer A replaces the duplication with the new abstraction.
   => Ah, the code is perfect. Programmer A trots happily away. Time passes.

4. A new requirement appears for which the current abstraction is almost perfect. Programmer B gets tasked to implement this requirement.
   => Programmer B feels honor-bound to retain the existing abstraction, but since isn't exactly the same for every case, they alter the code to take a parameter, and then add logic to conditionally do the right thing based on the value of that parameter.
   => What was once a universal abstraction now behaves differently for different cases.

5. Another new requirement arrives. => Programmer X. Another additional parameter. Another new conditional.
   => Loop until code becomes incomprehensible.

6. YOU ARE HERE => Don't fall for the sunken-cost-fallacy and just use code duplication instead of wasting time!

More advice

There are a lot more good habits you can learn

Very brief summary of "Clean Code" by Robert C. Martin

Most of these things have room for interpretation and a lot is biased by personal preference.

Always question if someone tells you there is only a single way to do something without giving a good argument. Even if the person is (supposedly) more experienced.

wikipedia