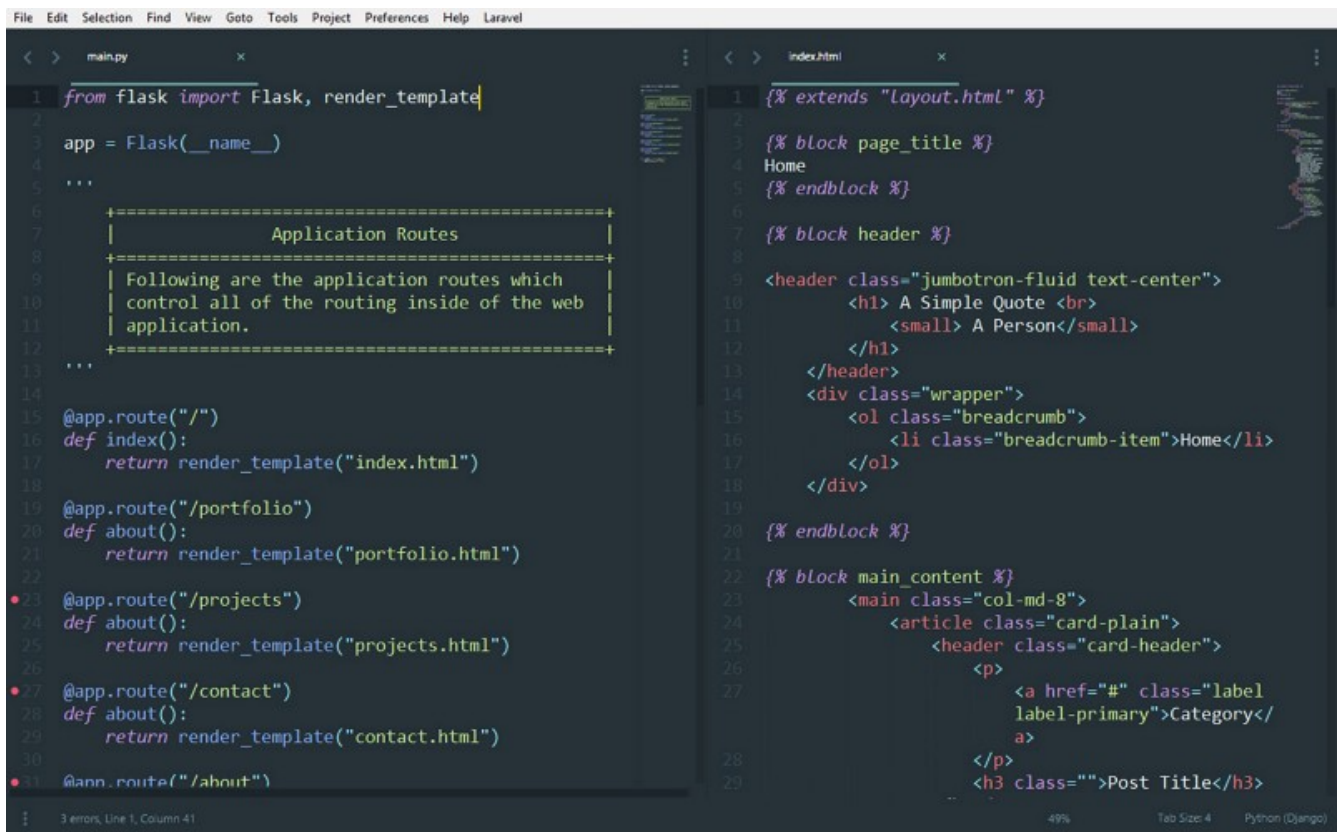


Basics of writing Clean code:

 medium.com/programming-hacks/basics-of-writing-clean-code-c1e79f3315d3



```
1 from flask import Flask, render_template
2
3 app = Flask(__name__)
4
5 '''
6     +=====+
7     |           Application Routes           |
8     +=====+
9     | Following are the application routes which |
10    | control all of the routing inside of the web |
11    | application.                               |
12    +=====+
13    '''
14
15 @app.route("/")
16 def index():
17     return render_template("index.html")
18
19 @app.route("/portfolio")
20 def about():
21     return render_template("portfolio.html")
22
23 @app.route("/projects")
24 def about():
25     return render_template("projects.html")
26
27 @app.route("/contact")
28 def about():
29     return render_template("contact.html")
30
31 @app.route("/about")
```

```
1 {% extends "layout.html" %}
2
3 {% block page_title %}
4 Home
5 {% endblock %}
6
7 {% block header %}
8
9 <header class="jumbotron-fluid text-center">
10     <h1> A Simple Quote <br>
11         <small> A Person</small>
12     </h1>
13 </header>
14 <div class="wrapper">
15     <ol class="breadcrumb">
16         <li class="breadcrumb-item">Home</li>
17     </ol>
18 </div>
19
20 {% endblock %}
21
22 {% block main_content %}
23 <main class="col-md-8">
24     <article class="card-plain">
25         <header class="card-header">
26             <p>
27                 <a href="#" class="label
28                     label-primary">Category</
29                 a>
30             </p>
31             <h3 class="">Post Title</h3>
```

A python flask program. Source: Pexels

I recently read my first iteration of Clean Code, the famous book by the prolific Uncle bob martin and it was one of the most delightful and eye opening experiences of my life as a programmer. There were lots of amazing takeaways from that book and personally would *highly recommend* it to folks who want to improve their craft in coding.

Why this post?

Master programmers think of systems as stories to be told rather than programs to be written.—Robert C Martin.

Though it is not possible to summarize the entire book's learnings in a blog post, i wanted to list down some practices/learnings from the book that IMHO can improve the quality of the code that we write each and every day.

This blog is not an attempt to explain each and everyone of them (*For that, you as a reader can just pick up the awesome clean code book 😊 or google*) but more as a suitable way for you to **“Know! what you Don’t know, you don’t know”**

Well let’s get started, Shall we?

Universal principles to follow: 🌐

There are some principles which are **so** universal in nature and are found repeated in so many different blogs and articles that they deserve a special mention.

- Avoid duplication anywhere in code(Famously known as **DRY** principle — Don’t repeat yourself)
- Follow **SOLID** principles to write clean classes and well organized API’s
- Follow design patterns and its associated terms **if it fits the problem space that you are trying to solve**. This instantly makes **your**code much more accessible to fellow programmers
- Follow law of demeter to enable less coupling and more cohesion

Law of Demeter says that a method f of a class C should only call the methods of these:

- C
- An object created by f
- An object passed as an argument to f
- An object held in an instance variable of C

Follow **three laws of TDD**, and keep your **test code** as clean (or even more so) as production code.

First Law: You may not write production code until you have written a failing unit test.

Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

Third Law: You may not write more production code than is sufficient to pass the currently failing test.

Functions

A Function should do one thing only and do it **really** well

Certain tips for writing effective functions:

- Avoid passing boolean into a function, this is a hint that func has an **if** statement within which causes it to do more than one thing.
- Functions should either do something or answer something, but not both. This ensures a function does not have hidden side effects. e.g a func named **isPresent()** should only return a bool and not do any other operations
- **Prefer** Exceptions to Returning Error Codes and extract error handling **try catch** into their own function.
- Avoid output arguments. Function if it has to, should change state of its owning object
- Code should always be separated with blank line to club logical blocks together. Think of different lines of code as **thoughts** and then always think of organizing similar thoughts together
- Each function should read like a newspaper, every functions implementation following its call and **having less vertical density**
- Don't return null

Object and data structures:

- Variables should be **private** so that we can change their type or implementation when required. There is no need to add getters/setter to each variable to expose them as public.
- Hiding implementation is **not just a matter of putting a layer of functions between the variables**. Hiding implementation is about **abstractions**! We do not want to expose details of data but rather express data as **abstract terms**

Exception handling:

- Try and extract **try catch** blocks into separate well named functions.

Having them mixed with other code just confuses the structure of the program. This is inline with “Function should do one thing”, Well error handling is one thing. 😊

- Prefer returning Exceptions instead of Error Codes.
- Each exception that you throw should provide enough context to determine the source and location of an error.

Variables:

- Variables should be declared as close to their usage as possible.
- Keep variables private and only expose **necessary** interactions as well defined **abstractions**. Avoid senseless getters and setters which expose all variables unnecessarily

Comments

| The only truly good comment is the comment you found a way **not to write**.

- Don't Use a Comment When You Can Use a **well named** Function or a Variable
- Any comment which forces you to look into another module for meaning has failed miserably in communicating and is not worth it at all.
- *Don't comment bad code, Rewrite it.* — Brian W. Kernighan and P. J. Plaugher

Boundaries

Always wrap third party code/API to minimize your dependency on it and allow the freedom to move to a different one in future without changing the consuming code.

Code organization and Design:

- Nearly all code is read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a **complete thought**. Those thoughts should be **separated from each other with blank lines**
- Local variables should be declared as close to their usage as

possible.



- Instance variables should be declared at top of the class since in a well defined class they would be used by multiple functions
- If one function calls another, they should be vertically close, and the **caller should be above the callee**, if at all possible. This gives the program a natural flow.
- Try to follow the **“The Principle of Least Surprise”** any function or class **should implement the behaviors** that another programmer could reasonably expect.
- It is **NOT** necessary to do a **Big Design Up Front(BDUF)**. In fact, BDUF is even harmful because it inhibits adapting to change, due to the ***psychological resistance to discarding prior effort*** and because of the way architecture choices influence subsequent thinking about the design.

According to Kent Beck, a design is “simple” if it follows these rules (In order of importance:

- Runs all the tests
- Contains no duplication
- Expresses the intent of the programmer
- Minimizes the number of classes and methods

Tests

Clean tests should follow F.I.R.S.T principles:

- **Fast:** ► Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently
- **Independent:**  Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like.
- **Repeatable:**  Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train

without a network.

- **Self-Validating:** ✓ The tests should have a boolean output. Either they pass or fail. You should not have to read through a log file to tell whether the tests pass.
- **Timely:** ⌚ The tests need to be written in a timely fashion.

Apart from this:

Ensure you write tests **just for a single concept**. Doing too many things in a test is usually a sign that it needs to be broken down.