Clean Code Developer Cheat Sheet

principles:

1. Don´t Repeat Yourself (DRY) rot

- no repetition of code
- recognition of repetition to prevent repetition of errors
- clean up repetitions through refactoring

2. Keep it simple, stupid (KISS)

- clear and simple solution to problems
- prerequisite for convertibility of the code

```python
def request_exploit_data(self):
    default_headers = self.headers
    url_body = 'https://www.exploit-db.com/exploits/'
    url = url_body + str(self.ebd_id)
    r = requests.get(url, headers=default_headers)
    return r
```

```python
@staticmethod
def get_exploit_from_db(exploit):
    element = collection.find_one({'exploit_id': exploit})
    if element:
        return element, True
    else:
        return None, False
```

```python
def execute_method(self):
    module = importlib.import_module(self.typ)
    class_ = getattr(module, self.typ.capitalize())
    call = getattr(class_, self.method)(self.value)
    return call
```

- Pictures: examples for KISS-orientated methods

3. Source code conventions

- Conventions should help to make code easier to read and understand
- The focus here is not only on the convention but also on the consistent use of these conventions.

```python
def request_exploit_data(self):
    default_headers = self.headers
    url_body = 'https://www.exploit-db.com/exploits/'
    url = url_body + str(self.ebd_id)
    r = requests.get(url, headers=default_headers)
    return r
```

```python
def cve_to_db(year):
    cve = Cve(year=year)
    path = cve.request_cve_data(cve.year)
    results = cve.parse_cve_data(path, cve.year)
    logging_msg = cve.insert_into_db(results)
    return logging_msg
```

4. Single Level of Abstraction (SLA)

- only one level of abstraction should be used in each method to strengthen the readability of the code
- hierarchical structure of classes analogous to a newspaper article from low to high levels of granularity

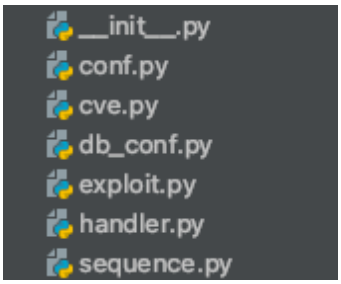
5. Single Responsibility Principle (SRP)

- one responsibility per class
- leads to a lower error rate when extending the functionality, since fewer classes have to be changed
- class CVE in cve.py for handling cve data
- class Exploit in exploit.py for handling exploit data
- class Handler in handler.py for executing methods, which are requested by user
- class Sequencer in sequence.py for handling user input and program flow
- physical and functional separation of responsibilities
- picture shows that class Exploit only has responsibility for exploit data

```python
class Exploit:
    def __init__(self, ebd_id):...

    def request_exploit_data(self):...



    @staticmethod
    def parse_exploit_data(request):...

    @staticmethod
    def entry_to_db(entry):
        collection.insert_one(entry)

    @staticmethod
    def exploit_to_db(exploit):
        exploit = Exploit(exploit)
        request = exploit.request_exploit_data()
        element = exploit.parse_exploit_data(request)

    @staticmethod
    def get_exploit_from_db(exploit):
        element = collection.find_one({'exploit_id': exploit})
        if element:
            return element, True
        else:
            return None, False
```

6. Separation of Concerns (SoC)

- functional separation of code units according topurpose
- class CVE in cve.py for handling cve data
- class Exploit in exploit.py for handling exploit data
- class Handler in handler.py for executing methods, which are requested by user
- class Sequencer in sequence.py for handling user input and program flow
- each code unit has a clearly defined task
- in class Exploit (picture above) you see the that the class Exploit only handles exploit functionality. This functionality is separated by concerns, e.g. one methods requests data, another one inserts data into the database



7. Interface Segregation Principle (ISP)

- Client or a program should not be dependent on parts of a service that it does not use
- The less details there are in the interface, the less coupling there is between the components, which reduces the susceptibility to errors.
-

8. Principle of Least Astonishment

- Software should be low in surprises. A test environment that tests changes and enhancements helps here. Functions should not change the state of the system and should always generate the same output with the same input.
- picture shows the function sequence, which handles the input of the user and controls the program flow. The function solves the problem recursive and does not change the state of the program. The same input always gives the same output. Changes in the databases are not executed by the Sequencer. These changes could create problems. Therefore these methods are separated and can be tested standalone.

```python
@staticmethod
def sequence(type='start', mode='default', test_type=None):
    seq = sequences[type]
    print(seq['text'])
    print(break_text)
    user_input = input(seq['input']) if mode == 'default' else test_type
    if mode == 'test':
        return
    if re.match(seq['pattern'], str(user_input)):
        if str(type) in cmd:
            print(user_input, seq['type'], seq['method'])
            Handler(method=seq['method'], typ=seq['type'], value=user_input)
            Sequencer.sequence(type='start')
        Sequencer.sequence(type=user_input)
    else:
        if str(user_input) == 'Start':
            Sequencer.sequence(type='start')
        elif str(user_input) == 'Exit':
            print('Thank you for using. Bye.')
        else:
            error_type = type + "_error"
            print(sequences[error_type]['text'])
            Sequencer.sequence(type=type)
```

9. Open Closed Principle

- classes should be able to be extended easily.
- Modifications should not be possible, since these are error-prone and thus the already implemented functions no longer function properly.

10. You Ain´t Gonna Need It (YAGNI)

- according to the YAGNI principle, the software requirements must be clearly defined and divided into small process steps
- this leads to the development of only necessary software parts and does not make the product unnecessarily large

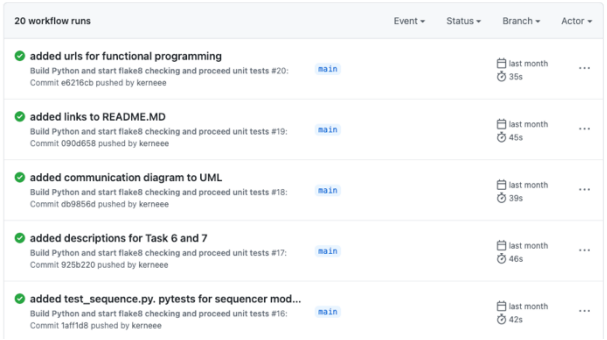
11. Law of Demeter

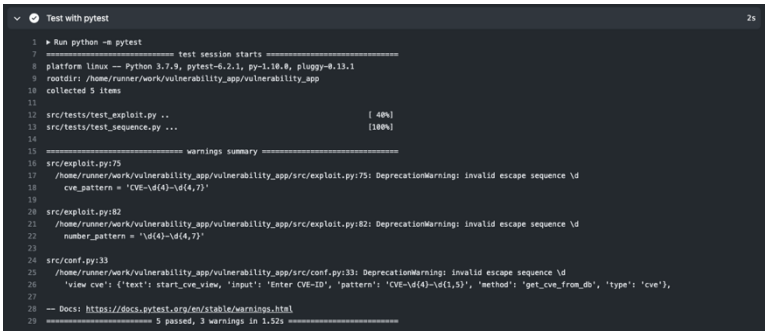
- Law of Demeter refers to the restriction of dependencies between classes.
- According to the principle, a class should only use methods of its own class, its own parameters, associated classes and self-generated objects.
- in some cases, pure data classes can make sense

practices:

1. Automated integration testing

- Testing code after change, automated and in the context of a version control system
- Tests check the code for errors and serve as a preliminary stage for operational use to guarantee efficient code development
- picture on the left GitHubAction Workflows with integrated pytest, picture on the right output of pytest



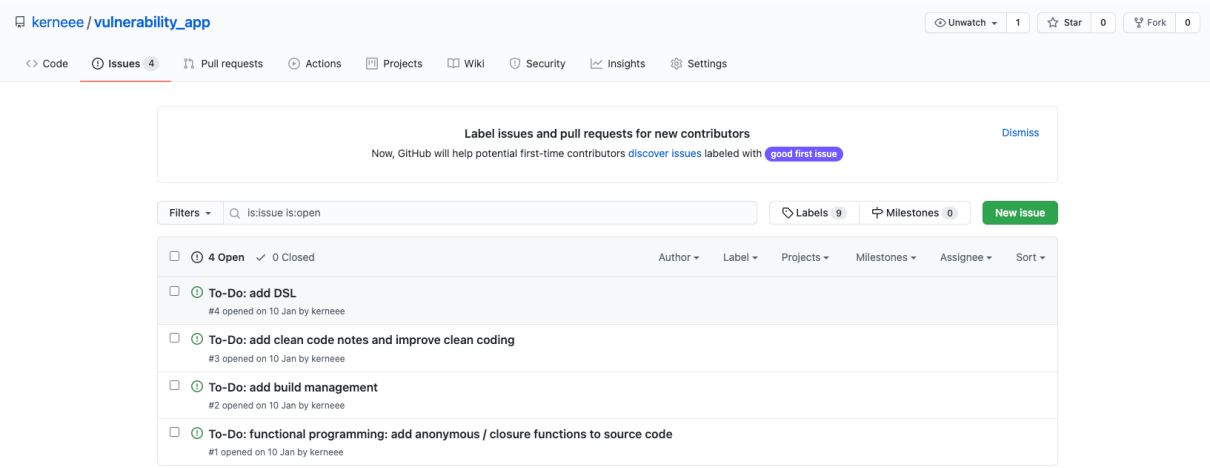


2. Continuous Integration

- see point 1 Automated integration testing for an example

3. Issue Tracking

- documentation of issues is necessary for an overview, to have, to prioritize and delegate them.



- You can see in the picture the use of the issue board in my git repository.

4. Code Coverage analysis

- The code coverage analysis provides information on the proportion of software parts secured by unit tests.
- Here, 100% code coverage cannot always be achieved with reasonable effort, but values over 90% are absolutely necessary.