Clean code practise with Python

Variables

Use meaningful and pronounceable variable names

Bad code:

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

Good code:

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

Use the same vocabulary for the same type of variable

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

```
get_user_info()
get_client_data()
get_customer_record()
```

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

```
get_user_info()
get_user_data()
get_user_record()
```

Use searchable names

It's important that the code we do write is readable and searchable. By *not* naming variables that end up being meaningful for understanding our program, we hurt our readers. Make your names searchable.

Bad code:

```
# What the heck is 86400 for?
time.sleep(86400);
```

Good code:

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

time.sleep(SECONDS_IN_A_DAY)
```

Bad code:

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

save_city_zip_code(matches[1], matches[2])
```

Good code:

Decrease dependence on regex by naming subpatterns.

```
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)

save_city_zip_code(matches['city'], matches['zip_code'])
```

Avoid Mental Mapping

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

Bad code:

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

for item in seq:
    do_stuff()
    do_some_other_stuff()
    # ...
    # Wait, what's `item` for again?
    dispatch(item)
```

Good code:

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

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

Don't add unneeded context

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

Bad code:

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

Good code:

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

Use default arguments instead of short circuiting or conditionals

Good code:

```
def create_micro_brewery(name: str = "Hipster Brew Co."):
    slug = hashlib.sha1(name.encode()).hexdigest()
    # etc.
```

Functions

Function arguments (2 or fewer ideally)

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

Zero arguments is the ideal case. One or two arguments is ok, and three should be avoided. Anything more than that should be consolidated. Usually, if you have more than two 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 code:

```
def create_menu(title, body, button_text, cancellable):
    # ...
```

Good code:

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

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

Functions should do one thing

This is by far the most important rule in software engineering. When functions do more than one thing, they are harder to compose, test, and reason about. When you can isolate a function to just one 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 code:

```
def email_clients(clients: List[Client]):
    """Filter active clients and send them an email.
    """
    for client in clients:
        if client.active:
            email(client)
```

Good code:

```
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 clients:
        email(client)
```

Function names should say what they do

Bad code:

```
class Email:
    def handle(self) -> None:
        # Do something...

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

Good code:

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

Functions should only be one level of abstraction

When you have more than one level of abstraction, your function is usually doing too much. Splitting up functions leads to reusability and easier testing.

Bad code:

```
def parse_better_js_alternative(code: str) -> None:
    regexes = [
        # ...
    ]

    statements = regexes.split()
    tokens = []
    for regex in regexes:
        for statement in statements:
            # ...

    ast = []
    for token in tokens:
        # Lex.

    for node in ast:
        # Parse.
```

Good code:

```
def parse_better_js_alternative(code: str) -> None:
    tokens = tokenize(code)
    syntax_tree = parse(tokens)

    for node in syntax_tree:
        # Parse.


def tokenize(code: str) -> list:
    REGEXES = [
        # ...
    ]

    statements = code.split()
    tokens = []
    for regex in REGEXES:
        for statement in statements:
           # Append the statement to tokens.

    return tokens


def parse(tokens: list) -> list:
    syntax_tree = []
    for token in tokens:
        # Append the parsed token to the syntax tree.

    return syntax_tree
```

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 code:

```
from pathlib import Path

def create_file(name: str, temp: bool) -> None:
    if temp:
        Path('./temp/' + name).touch()
    else:
        Path(name).touch()
```

Good code:

```
from pathlib import Path

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

def create_temp_file(name: str) -> None:
    Path('./temp/' + name).touch()
```

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, or accidentally wiring all your money to a stranger.

Now, you do need to have side effects in a program on occasion - for example, like in the previous example, you might need to write to a file. In these cases, you should centralize and indicate where you are incorporating side effects. Don't have several functions and classes that write to a particular file - rather, have one (and only one) service that does it.

The main point is to avoid common pitfalls like sharing state between objects without any structure, using mutable data types that can be written to by anything, or using an instance of a class, and not centralizing where your side effects occur. If you can do this, you will be happier than the vast majority of other programmers.

Bad code:

```
# Global variable referenced by following function.
# If another function used this name, now it'd be an array and could break.
name = 'Ryan McDermott'

def split_into_first_and_last_name() -> None:
    global name
    name = name.split()

split_into_first_and_last_name()

print(name)  # ['Ryan', 'McDermott']
```

Good code:

```
def split_into_first_and_last_name(name: str) -> None:
    return name.split()

name = 'Ryan McDermott'
new_name = split_into_first_and_last_name(name)

print(name)      # 'Ryan McDermott'
print(new_name)  # ['Ryan', 'McDermott']
```

Resources: 1. https://bit.ly/3oiEWpy

2. https://bit.ly/3D4wHnd