Clean_code

September 17, 2024

1 Clean code practices

1.0.1 General

Functions and methods should have verbs as names. Active voice! - calculate_time() - radiate()

Classes and data variables should have nouns as names. - DosisCalculator - RadiationController

Short variable names, like i, j, x, y, z, dx can be used - if they are well known domain specific names **or** - if they go out of scope within 5 lines of code, and they don't carry an important concept

Modules, classes, functions and methods should have docstrings.

Don't litter your core namespaces. Make good use of modules. Avoid global data.

Code that doesn't have unit tests is assumed to be faulty.

Don't repeat yourself (DRY) Code should not be produced with copy/paste.

Repetitive code should be refactored to conform to DRY.

1.0.2 Variables

Use meaningful and pronounceable variable names Bad:

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

Good:

```
[ ]: 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:

```
[ ]: get_user_info()
     get_client_data()
     get_customer_record()
```

Good:

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()
```

Even better:

```
Python is (also) an object oriented programming language. If it makes sense, package the functi
```

```
class User:
    info : str

    @property
    def data(self) -> dict:
        pass  # ...

    def get_record(self) -> Union[Record, None]:
        pass  # ...
```

Use searchable names We will read more code than we will ever write. 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:

```
time.sleep(86400)
```

Good:

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

time.sleep(SECONDS_IN_A_DAY)
```

Use explanatory variables Bad:

```
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])
```

Not bad:

```
It's better, but we are still heavily dependent on regex.
```

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

city, zip_code = matches.groups()
```

```
save_city_zip_code(city, zip_code)
```

Good:

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:

[ ]:
```
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:

[ ]:
```
locations = ('Eslöv', 'Skånings Åsaka', '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:

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

Good:

[ ]:
```
class Car:
    make: str
```

```
    model: str
    color: str
```

Use default arguments instead of short circuiting or conditionals Why write:

```
[ ]: def create_micro_brewery(name):
         name = "Hipster Brew Co." if name is None else name
         slug = hashlib.sha1(name.encode()).hexdigest()
         # etc.
```

… when you can specify a default argument instead? This also makes it clear that you are expecting a string as the argument.

Good:

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

1.0.3 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:

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

Good:

```
[ ]: 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
```

```
    }
)
```

Also good:

[ ]:
```
class MenuConfig:
    """A configuration for the Menu.

    Attributes:
        title: The title of the Menu.
        body: The body of the Menu.
        button_text: The text for the button label.
        cancellable: Can it be cancelled?
    """
    title: str
    body: str
    button_text: str
    cancellable: bool = False


def create_menu(config: MenuConfig):
    title = config.title
    body = config.body
    # ...


config = MenuConfig
config.title = "My delicious menu"
config.body = "A description of the various items on the menu"
config.button_text = "Order now!"
# The instance attribute overrides the default class attribute.
config.cancellable = True

create_menu(config)
```

Fancy:

[ ]:
```
from typing import NamedTuple


class MenuConfig(NamedTuple):
    """A configuration for the Menu.

    Attributes:
        title: The title of the Menu.
        body: The body of the Menu.
        button_text: The text for the button label.
        cancellable: Can it be cancelled?
```

```
    """
    title: str
    body: str
    button_text: str
    cancellable: bool = False


def create_menu(config: MenuConfig):
    title, body, button_text, cancellable = config
    # ...


create_menu(
    MenuConfig(
        title="My delicious menu",
        body="A description of the various items on the menu",
        button_text="Order now!"
    )
)
```

Even fancier:

[ ]:
```
from dataclasses import astuple, dataclass

@dataclass
class MenuConfig:
    """A configuration for the Menu.

    Attributes:
        title: The title of the Menu.
        body: The body of the Menu.
        button_text: The text for the button label.
        cancellable: Can it be cancelled?
    """
    title: str
    body: str
    button_text: str
    cancellable: bool = False

def create_menu(config: MenuConfig):
    title, body, button_text, cancellable = astuple(config)
    # ...


create_menu(
    MenuConfig(
        title="My delicious menu",
```

```
        body="A description of the various items on the menu",
        button_text="Order now!"
    )
)
```

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:

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

Good:

[ ]:
```
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)
```

Do you see an opportunity for using generators now?

Even better:

[ ]:
```
def active_clients(clients: List[Client]) -> Generator[Client]:
    """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 clients:
        email(client)
```

Function names should say what they do Bad:

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

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

Good:

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

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

Functions should only be in 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:

[ ]:
```
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:

[ ]:
```
REGEXES = (
   # ...
)

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:
    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:

[ ]:
```
from pathlib import Path

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

Good:

[ ]:
```
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 *pure function* will always return the same result given the same input. It will in no way affect any other system state.

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 firing all your nuclear missiles.

You need to have side effects in a program on occasion - keyboard/mouse input - screen output - file I/O - socket I/O - read system time - talk to subprocesses

In all these cases, centralize the functionality. Here, Logical Cohesion makes sense.

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:

[ ]:
```
# This is a module-level name.
# It's good practice to define these as immutable values, such as a string.
# However...
name = '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 name
    name = name.split()

split_into_first_and_last_name()

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

OK. It worked the first time, but what will happen if we call the function again?

Good:

[ ]:
```
def split_into_first_and_last_name(name: str) -> list:
    return name.split()

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

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

Also good:

[ ]:
```
from dataclasses import dataclass

@dataclass
class Person:
    name: str

    @property
```

```
    def name_as_first_and_last(self) -> list:
        return self.name.split()

# The reason why we create instances of classes is to manage state!
person = Person('Ryan McDermott')
print(person.name)  # 'Ryan McDermott'
print(person.name_as_first_and_last)  # ['Ryan', 'McDermott']
```

1.0.4 Classes

SOLID - Single responsibility - Open–closed - Liskov substitution - Interface segregation - Dependency inversion

Single Responsibility Principle (SRP) Every module, class, or function should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or function.

All its services should be narrowly aligned with that responsibility.

A class should have only one reason to change

Open/Closed Principle (OCP) Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Liskov Substitution Principle (LSP) If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.).

More formally, the Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called (strong) behavioral subtyping, that was initially introduced by Barbara Liskov in a 1987 conference keynote address titled Data abstraction and hierarchy.

It is a semantic rather than merely syntactic relation, because it intends to guarantee *semantic interoperability* of types in a hierarchy, object types in particular.

Subtype Requirement:

- Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Interface Segregation Principle (ISP) No client should be forced to depend on methods it does not use.

ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called *role interfaces*.

ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

Dependency Inversion Principle (DIP) A specific form of decoupling software modules.

When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details.

The principle states:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).
2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

By dictating that both high-level and low-level objects must depend on the same abstraction, this design principle inverts the way some people may think about object-oriented programming.

The idea behind points 1. and 2. of this principle is that when designing the interaction between a high-level module and a low-level one, the interaction should be thought of as an abstract interaction between them.

This not only has implications on the design of the high-level module, but also on the low-level one: the low-level one should be designed with the interaction in mind and it may be necessary to change its usage interface.