Python Guidelines

Release 0.1.0

Paul Wolf

May 31, 2022

CONTENTS:

This is a set of guidelines to make Python code more readable, easier to maintain and easier to reuse.

The concentration is on sound coding rather than big issues around architecture. We don't pay much attention to how to make code faster. We're more interested in the process of efficient code production and code maintenance. Writing idiomatic Python is a sound basis for efficient code.

If you struggle with too many bugs and maintanance problems in your project, applying the following recommendations might have a big positive effect. The intent is to bring together many well-known good practices in a checklist form to provide a toolkit for code reviews. This is not an authoritative set of prescriptions. It's a starting point for developing your own set of guidelines. Discard or modify practices as you see fit.

You'll need to be familiar with Python to some extent since we won't explain basics of the language. But there are helpful references in the appendix especially for common design patterns. If you are not familiar with some parts of Python like how modules and packages work, you'll need to read up in the Python Standard Library documentation.

Most of this material is applicable to most Python versions but we assume a more or less current version like 3.7 or even higher.

CHAPTER

ONE

TOOLS

1.1 Repls and debuggers

Every Python developer should be competent with these tools:

- ipython: This is the standard repl these days
- Jupyter Notebooks: a notebook that uses iPython at its core
- ipdb: the command line debugger of iPython or some pdb variant.

It should be easy - very easy - to load code into each of these tools not just load into *your* IDE's debugger or repl. If it's hard to do this, it's usually because context-dependent variables (see below) are being passed excessively in the call graph making setup for the debugging session complicated and time intensive. It will also make writing unit tests a lot more difficult.

For the purposes of this article, it is irrelevant what code editor you use, VSCode, PyCharm, Sublime, VIM, Emacs, etc. But be aware that just because it's easy to jump between definitions and usages of a function, class, method, etc. does not mean your code is well structured. Don't make familiarity with your IDE an assumption in the code design.

1.2 Code Improvement Utilities

Some popular tools provide the best indication of things to fix or even fix things automatically on your behalf. This is the easy part. Run these tools always before pushing code:

- `black` https://github.com/psf/black: you run this and agree that everyone in the team follows the style laid down by `black`. It is the basis for applying other tools mentioned below. Always run this first because it will fix a ton of things that would otherwise be flagged by `flake8`.
- `flake8` https://flake8.pycqa.org/en/latest/: This tool wraps three other tools. The best thing about it is the defaults are immediately useful. Run this and fix *every* raised issue. You can configure it to skip some checks but mostly skipping checks is useful only for an exisiting code base. For new code, it is important to not play with the default for function complexity before pushing code. `flake8` wraps other tools and has default settings that let you use it with minimal configuration effort for a big return on investment.

If you don't already run these tools, your code will experience a massive improvement after fixing issues identified by `flake8`.

Other tools you should consider:

- `` `mypy `` <http://mypy-lang.org/>`__: this tool will find bugs but also forces you to not do things that work but which are bad practices, like having functions that return unexpected types. But it also improves readability massively, IMHO. Fix *everything* flagged by `mypy`. `mypy` is useless unless you use type hints in your code. While there is vigorous debate about the benefits of type hints, I personally find them unquestionably useful when used appropriately. If you have no type hints, running `mypy` will find no problems. If you have type hints but never run `mypy` (or one of the other type checkers), you will find many problems upon finally doing so. Better to run it consistently after adding your first type hint. Fix every raised issue.

- `` `pylint `` <https://www.pylint.org/>`__: This is a great tool and should be used on any significant project. But configuration is non-trivial. At first you will get more out of `flake8` plus `mypy`. You can start using `pylint` and gradually build a configuration that works for you.

You'll notice, I don't talk about line length or how to format comprehensions or imports or other style issues. That's because you are running `black` and that tool decides for you. Style preferences of individual programmers creates unneeded scanning overhead that you can get rid of instantly with `black`.

One thing you should definitely not do is use type hints and then never run `mypy`. Why? Because your type hints will be wrong. This is wrong:

```python
def foo(a) -> str:
    try:
        return bar(a)
    except Exception:
        return None
```

The type hint should say `Optional[str]` because it might return `None`. If this kind of thing accumulates, you have a mess on your hands. If you use type hints, you need to run `mypy` and fix everything every time. This will not be onerous at all if you are consistent.

Use https://pre-commit.com/ to run tools automatically before a `git commit`.

CHAPTER

TWO

BASIC PYTHON CODE GUIDELINES

Code reuseability and maintenance is about managing complexity. We don't care how hard machines have to work to understand things as long as time/space tolerances are observed. We want our code to be readable by human beings. Source code complexity is about the ability of other developers to understand your code. Scanning is the process of reading code to understand its consequences. "What is this code doing?" means what side effects does it have, what data does it produce given specific inputs or, maybe most often, what is wrong with it? Complexity causes the reader cognitive load and consequently these questions are hard to answer. Recommendations in this article are intended to make scanning easier and to progressively implement code that can be managed, changed and fixed more easily.

2.1 Post hoc Refactoring vs Upfront Design Investment

It is a common sentiment that it's better to do the code right or not at all. I.e. front-load design effort. In contrast, Agile methods prescribe lower upfront design effort. It's probably only a slight distoration to suggest that Agile is based on the notion that the code will not be very good anyway.

My view is that every development team should devote at least 30% of scheduled time to refactoring. This does depend on what kind of project you are working on. If it's a marketing product web site that has a six month lifespan, you are not going to feel a burning desire to invest time into refactoring. But any framework code and long-term product code probably could use some level of refactoring at any given point. Failure to formally schedule refactoring time will lead to steady decline in maintainability of the code base as technical debt rises with no counter pressure.

2.2 Side Effects and State Management

The most important improvement you will ever achieve in your code is being clear about changing the state of your data. This affects the ability to reuse code and avoid bugs more than any other factor.

The functional paradigm is useful here. Implement functions without side-effects as far as possible. Give data via parameters to a function and get data back via return values.

Most applications need to have side-effects like sending messages, writing to databases, etc. But strive to understand where to use these operations without polluting your code with artefacts of your web framework or data storage framework. A web view function can and should call data storage apis. Just don't pass the objects that represent these APIs any more than required to other functions.

Python is not a functional programming language but you can apply functional programming principles to a useful extent. Much of what follows is about state management, the key to readable code, reuse and reducing errors in software. While functional purity is relatively easy to grasp for most developers, designing classes often causes good intentions to crumble. We present a set of guideines below to deal with class design complexity.

In the following, I prescribe various practices, not always with detailed explanations as to why. The preference is for conveying a set of recommendations along with a code review checklist. If you don't have that already, this provides you with something ready-to-go. If you don't like some of the recommendations, fork the checklist and modify to suits your needs and opinions.

Likewise, I don't give too many rules that are already covered by `black`, `flake8`, `pylint`. `black` is dictatorial by design. `flake8` will patiently explain.

What we do not say:

- Make your functions only x lines of code and no more?

This is poor advice. We want our functions to be only as complex as required but no simpler. Lines of code (LoC) does not equal complexity although LoC is part of any complexity measurement. `flake8` completely takes care of this for you. Don't worry about it. Just do what `flake8` says. You can configure it to adjust complexity threshold. Be careful about using that. There are cases where you can skip on an individual file or line basis. Use `#noqa` judiciously.

- Break up your classes to make them all small?

Again, no. Don't just reduce the size of classes because "smaller is better". Apply the below principles to achieve "as complex as required but no more." Classes become bloated when they are misused as pure namespaces. Encapsulation and information hiding are decent concepts but they are sometimes conflated with namespacing. It's more important to think in terms of how variables change over the lifetime of a class instance.

2.3 Dead code

Do not use your code base to store code that was once useful or which might be useful "one day". Ruthlessly root out unused code. There are utilities that can help with this, but you probably can browse your code base and find unused code easily enough.

2.4 Types

Use immutable types whenever you can:

- `tuple` instead of `list`: like `(1,2,3)`
- `frozenset` instead of `set`: `frozenset({1,2,3})`

Make sure you are not using dicts when you should be using another type, like one of these:

- namedtuple
- datasets
- Enum

In particular don't use dicts as enumeration types:

```
STATUS = {
    "READY": 0,
    "IN_PROGRESS": 1,
    "DONE": 2,
}
```

Use an Enum for this:

```
from enum import Enum
class Status(Enum):
    READY = 0
```

(continues on next page)

(continued from previous page)

```
    IN_PROGRESS = 1
    DONE = 2
```

You get better type checking, immutability and excellent `__repr__` output.

If you have built a mutable type, like a `list`, turn it into `frozenset` or `tuple` if it will be used later without requiring changes. If you are returning that type and don't need it to change, it is better to return the immutable type. If the user will change it, make them cast it to a mutable type. This will help readers to understand the developer intends to do state changes on that object.

For `dataclasses`, make sure you use `frozen=True`.

One day there will be a `frozenmap` type. But you probably want one of the above anyway. Using immutable types helps readability because the reader knows to scan past usages of immutable instance types searching for state changes.

2.5 Modules and Packages

Use modules and packages as namespaces. Import the module name preferably and call a function qualified by the module name. Now the the reader doesn't have to scroll to the top of the file to find out where the function comes from. If the function is unqualified, it's from the current module.

Python supports a feature to indicate protected and private names, where you prefix with either a single or double underscore. If you use this feature, you need to be consistent or it gets very confusing.

You generally want to use module namespaces to convey where things are coming from. You might want to hide some complexity by importing into `__init__.py`. The user of that package will then import those functions or classes without knowing the exact files where the implementation resides. This is not necessarily a good thing. It deprives the reader of the code of useful information.

There are diverging opinions about whether `__init__.py` should contain code. On balance, it's probably better to only have imports and not implementation code. On anything but very small projects, you will probably use `__init__.py` a lot for refactoring. It's better to therefore only have imports.

Be aware that modules and packages are often referred to interchangeably even in the PSL. It matters little. Technically a package is a directory and it has a file called `__init__.py`. What is important is you have these ways to control access:

- `__all__` in a module governs what is visible outside the module during importing with `import *`
- Single underscore _ or double underscore __ in front of a name governs visibility from outside the module under some circumstances
- What you put in the `__init__.py` governs what is visible outside the package

Note that "visible" and "accessible" are two different things. Python is not very rigorous about this. Since Python is a highly permissive language, rather then relying on some enforcement mechanism, make sure you adopt your own standards for importing modules.

More importantly, when you use these features, make sure you understand for whom you are using them: for the user of the module/class? Or for the reader of the module's code? Making your code intelligible to readers should be your highest priority.

2.6 Functions

Make functions pure in the functional programming sense, i.e. don't write functions with side effects when possible. Do not change the state of variables outside the function. But you can read data outside the function, like referencing module variables.

Avoid using closures and nested functions in general unless you have a compelling use case. Lambdas are too useful to avoid and generally can enhance readability if not misused. Don't assign a lambda expression to a variable; functions already have all the characteristics you need if you think you want that.

Brevity is not the defining criterion for a well-formed function. So, what is?

- Have a function do one well-defined thing.
- Have manageable state, as few variables as possible to achieve the single purpose of the function
- Make the function pure whenever possible
- Return immutable types whenever possible

You will sometimes update a mutable variable passed as a parameter (list, dict, etc.). The convention in Python is to return `None` if you update a list or dict passed to your function. So, that function has a side-effect. It's not pure. It is how some PSL (Python Standard Library) functions work like `sorted()` vs `list.sort()`.

But if you can, don't change the passed value. Return a new instance of an immutable type:

```python
from typing import Tuple, Sequence
import random

def remove_odd(data: Sequence[int]) -> Tuple[int]:
    return tuple(_ for _ in data if not _ % 2)

d = [random.randint(0, 100) for _ in range(10)]
even_data = remove_odd(d)
```

Now `even_data` is a tuple. This is good. To be clear, if you are changing the passed mutable variable, do not also return it.

Look for hanging indents that occur after `for` or `if` expressions. Very often if there are many lines of code under one of these, this block can be a separate function.

Reduce the number of separate variables given to a function or created by a function.

A good quick way to look for complexity is the number of indent changes. If you have many and variable indent changes in a function, you have more complexity. This plus LoC (lines of code) taken together gives an rough idea of complexity.`flake8` uses a formal complexity analysis tool but does not provide the sole indicator of complexity. But it is a great place to start. Reporting on complexity metrics in your CI pipeline is a great idea.

2.7 Default initialisations

Sticking to typical idioms in Python helps others read your code.

You could do this:

```python
def foo(default_list=None):
    if not default_list:
        default_list = list()
    ...
```

This is better, more idiomatic python:

```
def foo(default_list=None):
    default_list = default_list or list()
    ...
```

What you should not do:

```
# BAD
def foo(default_list=None):
    if not default_list: default_list = list()
    ...
```

It will work, but there is an idomatic way that is more expected.

If you need to change the value you'll need to use the more verbose conditional form:

```
# we want an int that is not zero or else None
user_id = int(user_id) if user_id else None
```

If `foo()` requires a list:

```
def foo(default_list: List):
    ...
```

You could call it like this if you think `my_list` might be `None`:

```
foo(my_list or list())
```

This is a feature of Python not shared with most other languages.

```
None or list()
```

will get you an empty list

```
list() or None
```

will result in None.

```
bool(list() or None)
```

will result in `False`.

2.7.1 Problems with `dict.get()`

Often, dicts are used to initialise a request or function call. They come as JSON and the developer makes use of the `get()` method to either get the provided value or supply a default. This is often done incorrectly.

What is the output of the following?

```
data = {"price": None}
float(data.get("price", 0))
```

Many developers will say `0`. If the "price" key is not present, the value of this expression is 0.0. All good. But because the call to `get()` in this case will return `None`, an exception will be thrown.

The solution is to parse and validate your input. If you wanted to use a dict nevertheless you are probably looking for this:

```
float(order.get("price") or 0)
```

A better solution would be to parse the incoming data into a dataclass. Dataclasses are now provided in Python and the PyDantic library provides parsing with validation.

2.8 Iterating

Use comprehensions instead of for loops where possible and appropriate.

This is verbose and hard to scan:

```
max_len = 0
for line in file:
    if line.strip():
        max_len = len(line) if len(line) > max_len else max_len
```

Compared to:

```
max(len(line) for line in file if line.strip())
```

This is brief and easier to scan. It does not require the use of a temporary variable, `max_len`, to hold state. It is a common idiom that a reader can rely on to expect no side-effects.

Another example:

```
filtered_events = list()
for event in events:
    if event.dt >= today and event.dt < tomorrow:
        filtered_events.append(event)
events = filtered_events
```

Compared to:

```
events = [e for e in events if e.dt >= today and e.dt < tomorrow]
```

Prefer the second one because the idiom generally promises no side effects whereas the `for` loop does not. The same goes for comprehensions. We do not expect side effects in a comprehension (or generator expression). The knowledge that there are no changes in the state of the program on the right side of the assignment is critical to our ability to mentally scan past that code when looking for state changes.

List comprehensions and higher order functions, `filter()`, `map()`, `reduce()`, etc., do nearly the same thing. Use list comprehensions by preference but don't worry if you prefer the higher order functions.

Functions you probably want to use that are not easily replaced with comprehensions:

- `zip()` https://docs.python.org/3/library/functions.html#all
- `all()` https://docs.python.org/3/library/functions.html#all
- `any()` https://docs.python.org/3/library/functions.html#any

Here's a hard-to-read prime number check function with *three* `return` statements that can be found frequently in the web:

```
def is_prime(x):
    if x >= 2:
        for y in range(2, x):
            if not ( x % y ):
                return False
    else:
        return False
    return True
```

Compared to one that is pythonic, easy to read and more correct:

```
def is_prime(n: int) -> bool:
    return all(n % i for i in range(2, n))
```

And, yes, the pythonic version is faster. You can produce side effects inside a comprehension but don't. Do not use comprehensions to loop through sequences without using the resulting sequence or collection (list, dict, etc.). If you only want the side effects of such an operation, use a `for` loop.

Gettting a tuple from a comprehesion is not quite consistent with other forms like dict and list comprehensions. You might think the following is a tuple comprehension:

```
e = (_ for _ in range(10))
```

But `e` is now a generator expression. Use this if you want a tuple right away:

```
e = tuple(_ for _ in range(10))
```

There are going to be times when you want to return a generator and not a tuple, like when the underlying data is large and requires iteration by the caller.

Use `dict.update()` instead of for loops to update a dictionary where possible or the merge `|` and update `|=` operators (from Python 3.9).

2.9 Initialisation

Most python developers know not to use a mutable default value in a function parameter declaration:

```
# BAD
def foo(my_list=[]):
    ...
```

While this does not result in catastrophe every time, you always want `my_list=None` and then make whatever changes are required to the logic in the function body. Also, when initialising in the body, use a callable instead of an empty list (`[]`):

```
my_list = list()
```

2.10 Comments and naming

The trend is towards fewer comments based on the assumption that other factors contribute to telling the reader what is going on. Especially eschew obvious comments. If you want to drive someone crazy do this:

```
# Bad
class Address:
    """This class represents an address."""
    ...
```

Follow the rule that if you have nothing useful to say, say nothing at all.

Assume you are writing your docstrings and comments first and then writing the code that implements what is described. You should name things - variables and functions - so that you can start removing the comments as the code becomes sufficiently readable that the comments do not add useful information. Remove any comment that does not add useful information.

Name variables in a more descriptive way the further they are used from their first use. If you are looping and using an index:

```
for i, name in enumerate(my_list_of_names):
    ...
```

`i` is ok for me if it lasts for very few lines, like three. If there are more lines of code, you'd be better off doing something like this:

```
for name_index, name in enumerate(my_list_of_names):
    ...
```

Type hints are a better form of documentation. The convention for a function docstring is something like:

```
def splice_name(first, last):
    """Return a str representing fullname."""
    return f"{first} {last}"
```

But now you can write:

```
def splice_name(first, last) -> str:
    """Combine first and last with space inbetween."""
    return f"{first} {last}"
```

Add more type annotations as necessary. Add a docstring unless it is immediately obvious what the function does. But don't bother identifying the return value type in the docstring if you already use a type hint for this purpose.

Now look what happens in iPython if I press `return` using `?` after the function name:

```
In [29]: splice_name?
Signature: splice_name(first, last) -> str
Docstring: Combine first and last with space inbetween.
File:      ~/prj/<ipython-input-28-b0b71e899c5a>
Type:      function
```

Likewise if you type `help(splice_name)`. This is amazingly useful.

2.11 Profiling code

Profiling code should not become a heavy source of technical debt. If a significant amount of code is just for profiling, this needs to be removed before production deployment. It's ok to leave in some code for timings, but it should be minimal. If you are leaving in too much profiling code, there is some fundamental design problem.

2.12 Don't reinvent

Don't create utilities for things the PSL (Python Standard Library) already provides. Especially things in `collections`, `itertools`, `functools`. Developers have a tendency to start building small utilities especially for namespaces that already exist in the PSL. The PSL versions are better than yours.

2.13 Unit tests and Linters

Unit testing is a required part of modern software development. It exposes problems in areas that you think you have not changed, regressions. It tests your intent versus what the software actually does. It makes it vastly easier to check your work. Unit testing is indispensible.

But unit tests are hard work. Whereas running a linter is trivial. It would be really strange to expend significant time on unit tests (which you should do) and then not run a linter.

When you write mostly pure functions, it's easier - much easier - to write unit tests.

When you refactor functions to satisfy complexity thresholds, you are making writing unit tests easier.

Also, you should very probably be using Hypothesis.

CHAPTER

THREE

CONTEXT DEPENDENT VARIABLES

These variables are complex, like classes that manage some state in a way that might not be apparent to a user via these means:

- The state is not stable in the current context. It might change in ways that are hard to predict.
- The state is produced in the first place by means that require processes outside the scope of the current program. Ie. it's not obvious how the state was constructed. The variable is intangled with some integration.
- The variable is an instance of a class with dependencies on code outside the PSL

Examples of context dependent variables:

- Request objects in a web framework
- A message queue
- An ORM object that represents and holds state about a database query and implements "advanced" features like caching, etc.

In the first instance, a web request, you might change the state by accessing methods on the object. In addition, it can be difficult to follow the construction of this object. In the second case, the object's state could be changing while you are accessing it.

Avoid passing these variables as parameters any more than necessary to other functions. If you need to give, for instance, the user object of a Request to a subroutine, do not do this:

```
# BAD
permissions = get_permissions(request)
```

Better:

```
permissions = get_permissions(request.user)
```

Best:

```
permissions = get_permissions(request.user.id)
```

Likewise:

```
# BAD
formatted = format_message(queue)
```

Better:

```
formatted = format_message(queue.pop())
```

A good example is the Django `Request` class. It has a `body` attribute. If you call `.read()` or `.readline()` on the request object, these change the state of the `body` attribute. If you pass `request` to a function, a reader of that function will not be able to assume the state of the object. It is also much more difficult to construct test instances of a `Request` object than to construct a user object. You can experiment with calls to `get_permissions()` and `format_message()` more easily in a repl. You can also use them in a context that doesn't require a request at all like if you are building a command line interface to these functions.

Below we discuss the Parameter Consolidation class pattern. This is a simple data class, in the sense of the PSL `dataclasses` module. This is not context dependent. It is simple to intialise and simple to understand the lifecycle of its state. A dataclass or simple class of your own construction is easy to create.

When trying out or testing code, it is desirable to be able to load, say, a function and pass parameters to it without excessive preparation of data needed for arguments.

A practical way to check if a variable is context dependent: Can I define it in a repl like ipython or a jupyter notebook easily?

CHAPTER

FOUR

CLASS DESIGN

Class design is non-trivial. It's where developers start having the most problems. Signs of class design problems:

- Using base classes as utilities for subclasses
- Swiss army knife syndrome, the class does a variety of things
- Complex parameterisation of `__init__()`
- Too many class instance variables
- You are having to create too many subclasses

Classes are for managing state, not behaviour. If you have a class that only has behaviour, it should probably be a module with one exception: when you need to define an interface. But in general, don't use a class as a namespace for behaviour only. Modules already do that.

Follow these basic rules for classes:

- A class should do one thing only
- A subclass should be a more specific instance of a parent class
- Minimise use of inheritance
- Avoid where reasonably possible multiple inheritance
- Use composition, not inheritance to acquire capabilities
- Avoid class variables
- Don't define constants in classes
- Don't have class methods that don't access `cls` or `self`

This last point helps you reduce class size using a reasonable rule. Move methods that don't access class or instance data to the module level as functions. This way, other developers can see immediately that they don't access or modify class state. If you think that method is part of the interface of the class, there is probably a design error since classes are used to manage state.

As mentioned above, use a file/module if all you want is a namespace for behaviour. If you have state to manage, then a class might be appropriate.

Any class state that does not change should be either a class variable or module level variable, preferably a constant. But consider moving that class constant outside of the class, since while applying the rule above, you will reduce the number of unneeded class methods. Class methods always invite the need to scan for state changes in the class instance variables. This is cognitive overhead that you want to reduce.

It's really ok if your classes dissolve into a series of pure functions in a module. This is a good thing because it's easier to understand provided each function does not operate on the same data repeatedly. If the functions mostly work on the same data and it's awkward to make them be outside a class, maybe a class is better.

- Variables should become constants if they don't ever change and the value is known before runtime.
- Constants should be moved out of classes.
- Constants should be moved out of modules into their own module if they are part of a general convention or protocol in your application, especially if they are used by multiple packages since this will help avoid cyclical imports.

You may find yourself moving towards a package structure like this:

- constants.py
- factories.py
- models.py # not ORM models, but common data structures as datacalasses
- [domain].py # where `domain` is the name describing what you are doing

Factories help to reduce dependencies when using composition. Constants help define common protocols and remove unchanging state from classes. Models are declarations of domain objects that have no framework or integration dependencies.

4.1 SOLID principles

- Single Responsibility Principle (SRP)
- Open/Closed Principle
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

4.1.1 Single Responsibility Principle (SRP)

Your class should do one thing. Don't ask a class to do more than one of these things:

- Acquire state
- Persist state
- Send messages (like Websockets, emails, events, etc.)
- Render context-specific representations of data
- etc (i.e. not a complete list)

4.1.2 Open-Closed Principle

Open for extension but closed to modification. When you create a class, your users should not need to change it to add features or adapt to a very specific case. But they should be able to extend that class.

4.1.3 Liskov Substitution Principle

Sounds fancy, but you already know about this: if you have an abstract base class, your subclasses should act like that abstract class would act (if it were not abstract). Same for a non-abstract base class and children.

4.1.4 Interface Segregation Principle

It's harder to do this in Python because Python does not have an interface language feature. If you do define an interface via an Abstract Base Class, do not force every implementing class to implement functionality not relevant to them. This hardly applies to Python because a class should do only one thing and subclasses should do that one thing in a way that is specialised.

4.1.5 Dependency Inversion Principle (DIP)

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

"Abstractions should not depend on details. Details should depend on abstractions."

–Agile Software Development; Robert C. Martin; Prentice Hall, 2003

You don't want your business rules and entity definitions to reference or be dependent on implementation details, like persistence frameworks, IO mechanisms or other low level things.

In summary:

The SOLID principles are valid, but strive for simplicity and appreciate the costs entailed by abstractions. There is such as thing as too much abstraction.

Don't introduce a dependency injection library in your project if you can get away with adding and modifying entities in your view function.

Frameworks with Object Relational Mappers make it very hard to keep to this principle because they closely couple two things: low level persistence details and the high-level domain entities. Trying to apply DIP here can cause things to get complicated very fast, i.e. techncial debt could spiral out of control. You might need to live with something that is imperfect.

There are many things to consider in building class hierarchies. The most important thing is to keep things simple. Secondly, always consider when you use a language feature if you are doing it for the class user or for the reader of the class code. The latter should be prioritised.

Composition and inheritance as competing design patterns is one of the most important things you can learn about how to use classes in Python:

Brandon Rhodes' guide to design patterns

4.2 Class naming

When naming a class, avoid using the word "Base" as a prefix for the earliest ancestor. It's better to choose a name that expresses what the class is because that will help you concentrate on the purpose of the class and subclasses. If you have a "Fish" hierarchy, you would not say

```
BaseFish
    FinFish
    ShellFish
```

You want the base class to be called “Fish”. This also means you won’t need to rename your class later when you find out you want to make your Fish class hierarchy derive from “Animal”.

4.3 Abstract classes

Use the ABC class to create an abstract class. Then use the `@abstractmethod` decorator for your abstract methods. Make sure you make properties abstract when they should be abstract. Abstract properties are awkward in Python, but the following works.

```
from abc import ABC, abstractmethod

class A(ABC):
    @property
    @abstractmethod
    def a(self):
        pass

    @abstractmethod
    def b(self):
        pass

class B(A):
    a = 1

    def b(self):
        pass
```

Remember when creating and passing class types, Python won’t check type identity when operating on what purports to be a specific object type. As long as one class seems to have the same behaviour as another class, it all works out. That’s duck typing. This lets any type of object impersonate any other type as long as it supports the same methods and properties that are used in the code that is handling these type instances.

4.4 Class Initialisation

Remember when you start initialising class instance properties the reader will ask herself which one of these stateful properties will be modified during the lifetime of the object. You need to make this easy, not hard.

If you assign a `self.myvar`, the reader cannot be certain of what happens later with that variable. Therefore, don’t use instance properties for constants. If you have a “base_url” that won’t change and is not initialised from a parameter, define it at module level or class level (a class property vs instance property). Reducing the number of class instance variables, reduces complexity of the class.

If you have a variable at module level in all upper case, it seems like overkill to also type hint it with `Final`. But using the type hint `Final` when assigning class instance variables is incredibly useful.

```
class Fish:
    def __init__(self, base_url):
        self.base_url: Final = base_url # good to know!
```

We always want to assume that a class property will never change. This might not be the case but almost always will be. There is rarely a good case for mutable class variables. If you want a singleton class pattern, remember this pattern already exists in Python in the form of module level variables. Don’t implement it in a class.

If you are defining constants closely associated with a class, it is probably still better to define them in a `constants.py` file.

4.4.1 Parameter Consolidation

Often, you start with a reasonably simple initialisation:

```
def __init__(self, name):
    ...
```

and later, it gets more complicated:

```
def __init__(self, name, street, postcode, town, country):
    ...
```

You then end up having many class instance variables that you have to manage. This causes a reader to have to scan the code more intensively to find out which variables get changed and when. It significantly degrades scanability of your code. It is better to use a Parameter Consolidation variable:

```
class Person:
   name
   street
   postcode
   town
   country
```

Now we initialise the class like this:

```
m = SnailMail(Person(data))
```

Or, even better, create it with a factory function or class.

```
m = snail_mail_factory(person=None):
    return SnailMail(person or Person())
```

There should be no obscurity about how this is constructed or any danger of the state changing after being passed to `__init__()`. It must be immutable. Python `dataclasses` (https://docs.python.org/3/library/dataclasses.html) are ideal for this, or use the Pydantic package (https://pydantic-docs.helpmanual.io/).

4.5 Class Factoring

Let's assume we need a class or classes to represent an integration with a remote service. How many classes will we have? Let's assume we need to represent getting variations of a data type from the same endpoint.

What you should not do: create an abstract base class that is a service provider or utility for subclasses.

- Remoteclient: Separate out the acquisition of state into a class that does only that. You can have different versions that implement caching or other pure state management functions.
- DomainManager: A class that manages the state in the sense of implementing any business rules.
- Formatter: A class or module level function that implements transformations on the data to make it fit specific usage scenarios.

- Factory: A class or module level function that creates appropriate DomainManager subclass and injecting the appropriate RemoteClient

Then only the DomainManger gets subclassed for specific kinds of data. If you need to parameterise any of these with complex set of data, use a Parameter Consolidation class.

Again, don't have any `@staticmethods`. Have only `@classmethods` that access the `cls` variable. But most of these can probably be module level functions which makes it easier to read the code, since they will be pure functions and your class will be smaller.

`self` and `cls` parameter names are conventions, not keywords. Be aware if some developer is using a different convention.

See the sample project for more information:

https://github.com/paul-wolf/python_coding/tree/main/fish

CHAPTER

FIVE

WEB FRAMEWORKS

Web apps are a class of application with sterotypical problems. Many problems with code complexity come with mixing up the web framework objects with business logic and service code. One important sign this happens is passing context dependent variables that represent constructs of the web framework to subroutines. There are two kinds of context dependent variables typical of web frameworks:

- Request and Response objects
- ORM objects if the web framework has or supports a database abstraction framework

You want to minimise passing of these objects through the call graph of your own code.

The principle feature of a web framework is the view function. The only relationship you want your view function to have with other parts of your code is data. The view function passes data to other parts of your application and gets data back. Passing a user id (int) is better than passing a User object. But if the called code will immediately again use that to query the database for user information, you now have an unnecessary call to the database. But maybe you have the data already, so pass that (company id?), but not in the form of a context-dependent variable that may have side effects and couples lower layers to the web framework. There are various options here but most importantly, you want to divest your function calls of couplings to your ORM or web framework.

If you end up passing the Request or ORM objects throughout your own code, far downstream, it will have dire consequences for readability and maintaining the code base.

If your view function can call the database, apply some business logic, return results, you are good to go. Don't try to make the view function into an abstraction layer that hides everything else. If the functionality is too complex and would cause `flake8` to judge the functionality of the view function to be over the accepted threshold, maybe reduce the view function to be short, handling mostly view things and let subroutines do the complex things. But do not then start coupling your view with the underlying routines by passing context dependent variables representing framework features. The benchmark is that the subroutines should be callable by non-view code. They should be easy to setup, easy to load in ipdb and ipython and require as few imports as possible.

If you are tempted to implement abstractions that support the Dependency Inversion Principle (DIP), be mindful that this can be hard to do when many web frameworks ensure tight coupling between high-level modules (your domain entitites and rules) and low level modules like the persistence layer. Fighting against this to achieve a Clean Architecure can have a high cost.

CHAPTER SIX

SUMMARY

Look at other people's code a lot, open source projects and the Python Standard Library itself.

But remember the popularity of a Python open source project is not a guarantee of good code practices to be emulated in your own projects. Some popular projects implement questionable design practices. Don't be led astray.

This article only touches the surface on all the things you need to think about. You can never get a complete picture of good practices from a single source. You should also not trust a single source as authoritative. The only way to achieve some mastery over Python is studying the advice of many people, articles, presentations, videos, etc.

Use the associated code review checklist as a starting point. Modify it as you research and gather experience.

CHAPTER

SEVEN

CLASS DESIGN EXAMPLE

We take an example of an application that calls a service to get fisheries data. The following code should run fine. It will print out common names of species.

You can find all the source code for this chapter here.

```python
from urllib.parse import urljoin

import requests


class DummyWebSocket:
    def notify(self, name):
        print(f"{self.__class__.__name__}: {name}")


mwi = DummyWebSocket()

#  https://fishbaseapi.readme.io/docs


class Fish:
    FIELD_NAME = "GenName"

    def __init__(self, base_url):
        self.base_url = base_url
        self.name_template = "Common name: {}"

    def _get(self, path):
        """Return response from calling fish service."""
        return requests.get(urljoin(self.base_url, path))

    def format_data(self, name_data):
        return self.name_template.format(name_data)

    def get_common_names(self):
        response = []
        for data in self._get("genera").json()["data"]:
            response.append(self.format_data(data.get(self.FIELD_NAME)))
        return response

    def get_names_and_notify(self):
```

(continues on next page)

(continued from previous page)

```
        for name in self.get_common_names():
            mwi.notify(name)


if __name__ == "__main__":
    fish = Fish("https://fishbase.ropensci.org/")
    print(fish.get_names_and_notify())
```

It is a common way for developers to write the first version of a class. It has various features that give it the appearance of flexibility:

- It lets the caller customise the url, allowing the called service to be changed if required
- It has separate methods for getting data vs notifying the UI or formatting that data
- The field in the returned data can be customised easily

However, there are some big drawbacks:

- To write an application, we'd spend a lot of time inside this class, firstly studying it to determine its behaviour and then re-writing to accommodate different behaviours
- It has to be modified directly rather than extended to achieve different behaviour. That's bad (see SOLID).
- It does a *lot* of things: acquiring state, notifying the UI, formatting in a specific way

Any attempt to build on this class while retaining the basic structure will result in a mess of unreadable, hard-to-manage code.

7.1 Refactored Fisheries Example

We can refactor everything to be vastly more flexible and easier to maintain and especially easier to test. Firstly, we'd create files to hold code that separates concerns:

- notifiers.py: this has code to notify
- clients.py: this has code to get state data from local or remote sources
- interfaces.py: this defines object types we need to instantiate that follow the protocol we want
- factories.py: this creates objects according to protocols we want
- formatters.py: this formats data in ways we need for different application purposes
- fish.py: this holds domain manager objects

In our application source, `fisheries.py`, which might be a view function, we'll import all the tools we've created:

```
import notifiers
import clients
import factories
import formatters
import fish
```

Now we can start doing various application tasks. This code is the fastest changing, least essential code:

```
fish_genera = factories.fish_factory(fish.FishGenera, client=clients.FishClientFile())

# Iterate the genera and report to the user interface via web sockets
for data in fish_genera:
    notifiers.FishUINotifier().notify(
        formatters.get_formatted_fish(data, lambda f: f"This is the genus name: {f}")
    )
```

We do all the mixing and matching of tools here. See the example source code, which you can also run:

https://github.com/paul-wolf/python_coding/tree/main/fish

CHAPTER

EIGHT

PYTHON CODE REVIEW CHECKLIST

8.1 General

- Code is blackened with `black`
- `flake8` has been run with no errors
- `mypy` has been run with no errors
- Function complexity problems have been resolved using the *default* complexity index of `flake8`.
- Important core code can be loaded in iPython, ipdb easily.
- There is no dead code
- Comprehensions or generator expressions are used in place of for loops where appropriate
- Comprehensions and generator expressions produce state but they do not have side effects within the expression.
- Use `zip()`, `any()`, `all()`, etc. instead of for loops where appropriate
- Functions that take as parameters and mutate mutable variables don't return these variables. They return None.
- Return immutable copies of mutable types instead of mutating the instances themselves when mutable types are passed as parameters with the intention of returning a mutated version of that variable.
- Avoid method cascading on objects with methods that return `self`.
- Function and method parameters never use empty collection or sequence instances like list `[]` or dict `{}`. Instead they must use `None` to indicate missing input
- Variables in a function body are initialised with empty sequences or collections by callables, `list()`, `dict()`, instead of `[]`, `{}`, etc.
- Always use the `Final` type hint for class instance parameters that will not change.
- Context-dependent variables are not unnecessarily passed between functions or methods
- View functions either implement the business rules the view is repsonsible for or it passes data downstream to have this done by services and receives non-context dependent data back.
- View functions don't pass `request` to called functions
- Functions including class methods don't have too many local parameters or instance variables. Especially a class' `__init__()` should not have too many parameters.
- Profiling code is minimal
- Logging is the minimum required for production use
- There are no home-brewed solutions for things that already exist in the PSL

8.2 Imports and modules

- Imports are sorted by `isort` or according to some standard that is consistent within the team
- Import packages or modules to qualify the use of functions or classes so that unqualified function calls can be assumed to be to functions in the current module

8.3 Documentation

- Modules have docstrings
- Classes have docstrings unless their purpose is immediately obvious
- Methods and functions have docstrings
- Comments and docstrings add non-obvious and helpful information that is not already present in the naming of functions and variables

8.4 General Complexity

- Functions as complex as they need to be but no more (as defined by `flake8`'s default complexity threshold)
- Classes have only as many methods as required and have a simple hierarchy

8.5 Context Freedom

- All important functionality can be loaded easily in `ipython` without having to construct dummy requests, etc.
- All important functionality can be loaded in pdb (or a variant, ipdb, etc.)

8.6 Types

Have immutable types, tuple, frozenset, Enum, etc. been used in place of mutable types whenever possible?

8.7 Functions

Functions are pure wherever possible, i.e. they take input and provide a return value with no side-effects or reliance on hidden state.

8.8 Modules

- Module level variables do not take context-dependent values like connection clients to remote systems unless the client is used immediately for another module level variable and not used again

8.9 Classes

- Every class has a single well-defined purpose. That is, the class does not mix up different tasks, like remote state acquisition, web sockets notification, data formatting, etc.
- Classes manage state and do not just represent the encapsulation of behaviour
- All methods access either `cls` or `self` in the body. If a method does not access `cls` or `self`, it should be a function at module level.
- `@classmethod` is used in preference to `@staticmethod` but only if the method body accesses `cls` otherwise the method should be a module level function.
- Constants are declared at module level not in methods or class level
- Constants are always upper case
- Abstract classes are derived from abc: `from abc import ABC`
- Abstract methods use the `@abstractmethod` decorator
- Abstract class properties use both `@abstractmethod` and `@property` decorators
- Classes do not use multiple inheritance
- Classes do not use mixins (use composition instead) except in rare cases
- Class names do not use the word "Base" to signal they are the single ancestor, like "BaseWhatever"
- Decorators are not used to replace classes as a design pattern
- `__init__()` does not define too many local variables. Use the Parameter Consolidation pattern instead.
- A factory class or function at module level is used for complex class construction (see Design Patterns) to achieve composition
- Classes are not dynamically created from strings except where forward reference requires this

8.10 Design Patterns

- Do not use designs that cause a typical Python developer to have to learn new semantics that are unexpected in Python
- Classes primarily use composition in preference to inheritance
- Beyond a very small number of simple variables, a class' purpose is to acquire state for another class or it uses another class to acquire state in particular if the state is from a remote service.
- If you use the Context Parameter pattern, it is critical that the state of the context does not change after calling its `__init__()`, i.e. it should be immutable
- If a class' purpose is to represent an external integration, you probably want numerous classes to compose the service: RemoteDataClient, DomainManager, ContextManager, Factory, NotificationController, DomainResponse, DataFormatter, etc.

CHAPTER

NINE

INDICES AND TABLES

- search