Subclassing in Python Redux

2021-06-22

The conflict between subclassing and composition is as old as object-oriented programming. The latest crop of languages like Go or Rust prove that you don't need* subclassing *to successfully write code. But what's a* pragmatic *approach to subclassing in Python, specifically?

[This article has been translated to Chinese: 再谈 Python 中的继承]

Anybody who follows me long enough knows that I'm firmly in the *composition-over-inheritance* camp. However, **Python is designed in a way that you *can't* write idiomatic code without subclassing *sometimes***. My goal for this article is to meditate on the question when that *sometimes* is and untangle my gut feelings on the topic[1].

I realize this blog post is *long*. In fact, it's the longest piece of prose I've written since my thesis in 2006. Objectively, I should've split it up into *at least* three parts. It would be better for engagement (SEO! Social media! Clicks!) and it would make it more likely that people *actually read it to the end*.

But I want it to stand for itself. I want it to be a distilled essence of what I've learned over the years. Of course, *you* can feel free to take as many breaks as you like – this article isn't going anywhere!

Let's start with *nuance*. One of the reasons why many discussion on subclassing are frustratingly fruitless is that there isn't just one type of inheritance. As the wonderful article *Why inheritance never made any sense* explains[2], **there are *three* types that should never be mixed** – no matter how you feel about subclassing in general.

That makes it possible that three people argue against each other, each one being right in their own way, never finding common ground. We've all seen these discussions unfold.

In my experience – if used strictly separately – one is good, one is optional but useful, and one is bad. Most problems with subclassing stem from the fact that we try to use more than one type of subclassing at once – or from focusing the object design on the bad type.

In all cases **you sacrifice reading convenience for writing convenience**. That is not necessarily *bad*, because software design is all about trade-offs, and you can conclude that it's perfectly worth it in some instances. This is why I *don't* expect you to agree with everything that follows. But I hope to provoke some thoughts that will help you to make that decision in the future.

But now, without further ado, let's look at the three types. Starting with the bad one.

Type 1: Code Sharing

> *"Namespaces are one honking great idea – let's do more of those!"*
>
> — **Tim Peters**, *The Zen of Python*

Most criticism of subclassing comes from code sharing and rightfully so. I don't feel that I have a lot to add to it, so instead I'm going to link to exceptional prior art by people far smarter and more eloquent than me:

- *The Composition Over Inheritance Principle* by Brandon Rhodes,
- *The End Of Object Inheritance & The Beginning Of A New Modularity* by Augie Fackler and Nathaniel Manista at PyCon US 2013,
- and *Nothing is Something* by Sandi Metz at RailsConf 2015.

In a nutshell, there are three overarching problems:

1. **Variation over more than one axis.** This is the major practical takeaway from Brandon's post and the second half of Sandi's talk. It's not easily explained so I'll refer to their works, but the essence is: If you want to customize more than one behavioral aspect of a class, code sharing via subclassing won't work. It leads to *subclass explosion*.

This is not an opinion or a trade-off. *This* is a fact.

2. **Class and instance namespaces get muddled.** If you access an attribute `self.x` in a class that inherits from one or more base classes, you're telling the reader of your code: "There's an attribute `x` *somewhere* in the class hierarchy, but I won't tell you where. Good luck finding it!" It takes research and mental energy to find out where `x` is coming from. This is true while *reading code* and this is also true while *debugging*.

   It also means that there's always the danger of two classes in the same hierarchy – that don't know about each other – trying to have an attribute with the same name. Python has the concept of the double underscore prefix (`__x`) to deal with this scenario, but that has been frowned upon and argued to prefer principle of *consenting adults*.

   The problem is that **informed consent is impossible if all parties aren't informed**. This problem gets exponentially worse with multiple inheritance and its extreme form of mixins. You're relying on classes – that you potentially don't control and that know nothing about each other – getting along in a *shared namespace*.

   Another problem is that you have **no control over the methods and attributes that *you* expose from the base classes to *your* users**. They are just there, tarnishing your API surface. Potentially changing over time as your base classes evolve and add or rename methods and attributes. This is among the reason why *attrs* (and ultimately *dataclasses*) chose to use *class decorators* instead of subclassing: you have to be deliberate in what you attach to a class. It's impossible to accidentally leak something to all subclasses.

3. **Confusing indirections.** This is a special case of the previous problem and the main point of Augie's and Nathaniel's talk. If every method is on `self`, it's unclear where it's coming from when looking at the call. Unless you are *very* careful, every attempt at understanding control flow ends in a wild goose chase. Once multiple inheritance comes into play, you better read up on the MRO and `super()`. I think it's fair to say that something is amiss if a question that boils down to "what is `super()` even for?" gets almost 3,000 upvotes and more than 1,000 bookmarks on *Stack Overflow*.

   All of this gets extra problematic if you build APIs that require subclassing for either implementing or overwriting existing methods that get called from somewhere else. Both *Twisted* and *asyncio* have committed these sins in their `Protocol`[3] classes respectively and it scarred me forever. The most common problems are that it's complicated to find out which methods exist (especially in deep hierarchies like *Twisted*'s) and the often silent failure if you name your method subtly wrong and the base class doesn't find it.[4]

> *"'Also the subclassing based design was a huge mistake' is probably the most-commonly uttered sentence in programming."*
>
> — **Cory Benfield**, *Tweet*

* * *

I use subclassing for code sharing only if I need to bend the behavior of a class *that I don't control*. I consider it a less egregious type of *monkey patching*. Usually it's better to write an *Adapter*, *Facade*, *Proxy*, or *Decorator*, but there are cases where the amount of methods that you'd need to delegate make that cumbersome if you want to change only a small detail[5].

In any case, **don't** make it a central part of *your* design.

Type 2: Abstract Data Types aka Interfaces

Abstract Data Types (ADTs) are mainly for **tightening interface contracts**. You want to be able to say that you want an object with certain properties (attributes, methods) and don't care about the rest. In many languages they are called *interfaces* which sounds a lot less pretentious, which is why I will be using that term from now on.

Since Python is dynamically typed and type annotations are strictly optional, you **don't need formal interfaces**. However, it is very useful to have a way to explicitly define an interface that you require for a piece of code to function. And since the advent of type checkers like *Mypy*, they've become ***verified* API documentation**, which I find wonderful.

For example, if you want to write a function that receives objects with a `read()`

method, you would somehow define an interface `Reader` that has that method (the *how* will be explained in a minute) and use it like this:

```
def printer(r: Reader) -> None:
    print(r.read())

printer(FooReader())
```

Your `printer()` function doesn't care what `read()` is doing as long as it returns a string it can print. It can return a pre-defined string, read a file, or make a web API call. `printer()` doesn't care and your type checker will yell at you if you try to call any other method than `read()` on it.

* * *

The Python's standard library comes with *two* approaches to defining interfaces:

1. *Abstract base classes* (ABCs) are a less powerful version of *zope.interface* and work using *nominal subtyping*. They have been around since Python 2.6 and the standard library is full of them.

   Please note that not every abstract base class is also an abstract data type. Sometimes it's just an *incomplete* class that you're supposed to complete by subclassing it and implementing its abstract methods – not an interface. The distinction is not always 100% clear, though.

2. *Protocols* avoid subclassing by using *structural subtyping*. They have been added in Python 3.8, but *typing-extensions* make them available as far back as Python 3.5.

Nominal subtyping and *structural subtyping* are big words, but fortunately they are straightforward to explain.

Nominal Subtyping

Nominal subtyping means you have to *tell* the type system that your class is a subtype of an interface definition. ABCs usually do that via subclassing but you can use the `register()` method too.

This is how you would define the `Reader` interface from the introduction and mark `FooReader` and `BarReader` as implementations of it:

```
import abc

class Reader(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def read(self) -> str: ...

class FooReader(Reader):
    def read(self) -> str:
        return "foo"

class BarReader:
    def read(self) -> str:
        return "bar"

Reader.register(BarReader)

assert isinstance(FooReader(), Reader)
assert isinstance(BarReader(), Reader)
```

If `FooReader` didn't have a method called read, instantiation would fail *at runtime*. If you use the `register()` route as with `BarReader`, the interface is *not* verified at runtime and it becomes a (as the docs call it) "virtual subclass". This gives you the freedom to use more dynamic – or magical – means to provide the desired interface. Since `register()` takes the implementing object as its argument, you can use it as a class decorator and save yourself two empty lines.

In nominal subtyping, multiple inheritance is not only accepted but *encouraged*, because ideally no methods, no behaviors, are being inherited and hopelessly intermixed – only class identities are being compounded. A class can implement many different interfaces and the smaller an interface, the better.

*   *   *

One ostensible "upside" of using ABCs to define interfaces is that by subclassing them, you can smuggle in code sharing by adding regular methods to your abstract base class. But as mentioned at the beginning: mixing subclassing types is a bad idea. Code sharing via subclassing is a bad idea. Multiple inheritance makes it an extra bad idea.

To be fair, I have seen good uses of this pattern, but you have to be very judicious with your approach. An idiomatic case in Python is when you need to implement a whole bunch of dunder methods[6] based on other, well-defined behavior. A good example is `collections.UserDict`. It's not great for all the reasons mentioned, but it's a good trade-off within the constraints and culture of Python. However, in the example of `UserDict`, it *would* get problematic the moment you tried to tack on *more* behavior on your subclass than what is expected from a dict. Then, the problems from the section on code sharing by subclassing would could back in force. Give classes only one responsibility to avoid that.

Structural Subtyping

Structural subtyping is *duck typing* for types: if your class fulfills the constraints of a `Protocol`, it's automatically considered a subtype of it. Therefore, a class can implement *many* `Protocol`s from all kinds of packages without knowing about them!

By default, this only works with type checkers, but if you apply `typing.runtime_checkable()`, you can also perform `isinstance()` checks against them.

The example from the previous section would look like this:

```python
from typing import Protocol, runtime_checkable

@runtime_checkable
class Reader(Protocol):
    def read(self) -> str: ...

class FooReader:
    def read(self) -> str:
        return "foo"

assert isinstance(FooReader(), Reader)
```

As you can see: `FooReader` doesn't know that the `Reader` protocol exists at all!

* * *

What I *really* like about `Protocol`s is how it allows me to define what interface I *need* completely non-intrusively and that definition can live along with the *consumer* of the interface. That's great when you have different implementations of the same interface in the same code base. For example you could have an interface `MailSender` that sends an email in production but just prints it to the console in development.

Or if you only use a small subset of a third-party class and want to be explicit which subset that is. This is great (verified!) documentation and helps when implementing fakes for your tests.

For more details on `Protocol`s and *structural subtyping*, check out Glyph's *I Want A New Duck*.

While this type of subclassing is *mostly harmless*, you **don't need subclassing for abstract data types** in Python thanks to `typing.Protocol` and ABCs's `register()` methods.

Type 3: Specialization

So we've had one subclassing type that's harmful and one subclassing type that's unnecessary. Now we've reached the good type. In fact, you can't get around this kind of inheritance in Python even if you wanted. Unless you want to stop using `Exceptions`.

Interestingly, specialization is often misunderstood. Intuitively it's easy: if we say that a class B specializes base class A, we say that class B is A with *additional properties*. A dog is an animal. An A350 is a passenger airplane. They have all the properties of their base class(es) and *add* attributes, methods, or just a place in a hierarchy[7].

Despite this alluring simplicity, it's often used incorrectly. The most notorious mistake is saying that a square is a specialization of a rectangle, because geometrically, it *is* a special case. However, **a square is not a rectangle *plus more***.

You can't use a square everywhere you can use a rectangle, unless the code *knows* that it has to expect a square too[8]. If you can't interact with an object as if it's an instance of its base class, you're violating the *Liskov substitution principle*[9] and you can't write polymorphic code.

> If you look closely, you realize that interfaces from the previous section are a special case of specialization. You always specialize an generic API contract into something *concrete*! The key difference is that abstract data types are … well … abstract.

I find specialization useful when I'm trying to represent *data* that is strictly hierarchical.

For instance, imagine you want to represent email accounts as classes. They all share some data like their ID in the database and the address, but then – depending on the type of the account – they (can) have additional attributes. Importantly, these added attributes and methods change little to nothing of the existing ones. For example, a mailbox that stores emails on the server needs login information in form of a password hash. An account that accepts emails and only forwards them to an another email address does not[10].

You end up with the following four approaches.

Approach 1: Create a Dedicated Class for Each Case

These are the classes that you effectively want in the end:

```
class Mailbox:
    id: UUID
    addr: str
    pwd: str

class Forwarder:
    id: UUID
    addr: str
    targets: list[str]
```

The address type is encoded in the class and each class has only the fields it uses.

And if your model is this simple, this would *absolutely* be the way to go. Any attempts at de-duplication only make sense if you have *many* more fields and more types.

Any method that you'd add to either class would be completely independent from the other – leaving no room for confusion. You can also use these classes with a type checker using the Union type: `Mailbox | Forwarder`.

* * *

Usually it's a good idea to start with this approach *in any case* because *duplication is far cheaper than the wrong abstraction*. Seeing all possible fields in front of you makes further design decisions a lot easier.

Approach 2: Create One Class, Make Fields Optional

“Conditions always get worse. Conditions reproduce.”

— **Sandi Metz**, *Nothing is Something*

This is a design that you might end up with when you try to avoid inheritance at all cost, but still avoid repeating yourself:

```
class AddrType(enum.Enum):
    MAILBOX = "mailbox"
    FORWARDER = "forwarder"

class EmailAddr:
    type: AddrType
    id: UUID
    addr: str

    # Only useful if type == AddrType.MAILBOX
    pwd: str | None
    # Only useful if type == AddrType.FORWARDER
    target: list[str] | None
```

Technically, this is more *DRY*, but it makes the *usage* of the instances of the class a lot more awkward. The type/existence of most fields entirely depend on the value of the `type` field, which only exists because all address types share the same class type.

It contradicts my favorite design principle to make illegal state unrepresentable and is impossible to sensibly check using a type checker, which would complain about accessing `None`-able fields all the time.

The fact that all behavior working on this class would be lumped together leads to a lot of conditions (`if`-`elif`-`else` statements) that increase the complexity of your code significantly. The whole point of polymorphism is to avoid that.

Having optional attributes[11] is potentially a red flag. Having fields that need a *comment* to explain *when* to use them is a May Day Rally. As controversial type annotations are, in this case they clearly point out to you that there's a problem with your models. Without them, you'd have to notice that your code is more complex than it should be, which isn't as straight-forward.

* * *

You can make the situation a bit less painful and move mailbox-specific data into a class and make just *that field* optional. It's better but still unnecessarily clunky.

Approach 3: Composition

This approach inverts the last one and looks silly with our overly simplistic data model, but let's just pretend that `EmailAddr` has many more fields such that it's worth being wrapped into an own class:

```
class EmailAddr:
    id: UUID
    addr: str

class Mailbox:
    email: EmailAddr
    pwd: str

class Forwarder:
    email: EmailAddr
    targets: list[str]
```

This approach is not that bad! We've got no optional fields and all data relationships are clear. As readability and clarity goes, there's nothing to complain about.

Except that it's also very clunky and you don't need to consult Guido to realize that it's everything but Pythonic. So why does it look so contrived, although composition is supposed to be better than inheritance? `EmailAddr` and `Mailbox` / `Forwarder` are *too closely related* – it's even awkward to name the field to store it. Composition isn't failing us, but *in this case* forcing an has-a relationship feels like going against the grain.

But it's useful to show us something about our models: they all have common base information and are closely related. So let's go the final step and use Python's way of sharing a common base and use specialization by subclassing. We will come back to composition when I improve the design of a subclassing-based design in a later section of this article.

Approach 4: Create a Common Base Class, Then Specialize

Finally, the approach that's in my opinion most ergonomic, DRY, obvious, and feasible to type check:

```
class EmailAddr:
    id: UUID
    addr: str

class Mailbox(EmailAddr):
    pwd: str

class Forwarder(EmailAddr):
    targets: list[str]
```

Whenever you have a `Mailbox`, you *know* you have a `pwd` field – and so do your type checkers. The type is encoded in the class, so you don't have to repeat it in a field. A `Mailbox` strictly **is** an `EmailAddr` *plus more*.

As for code, you now have to be aware of the **rules of responsible subclassing** like the aforementioned *Liskov substitution principle*. This *is* additional complexity and mental overhead, but the boundaries and responsibilities are much clearer.

> Subclassing *requires* knowledge and discipline *from* you. Composition *mechanically forces* discipline *on* you – even if it results in clunkiness.
>
> This is probably the simplest reason to err on the side of composition: **it leaves less room for errors from *you***.

Reading clarity suffers as with all kinds of subclassing, because you have to assemble the final class in your head to know what fields exist. But effectively you get the same classes like in the first approach. As long as you don't overdo it and ideally keep the definitions physically close to each other, it's the best trade-off in situations like this.

It's so useful that I've used it in my parsing library for PEM files and have yet to regret it.

*   *   *

A general advice to derive from this section is to always focus on the *shape* of your data *first* and only *then* what to do with it.

Once you have the shape nailed down, the behavior comes much more naturally. A good example of that is the *Sans I/O* movement that is unequivocally *data-first*, since the behavior is supposed to be replaceable *by design*.

As long as you avoid cross-hierarchy interactions between methods while specializing, you should be fine. But always ask yourself if a **function** wouldn't be enough – *especially* if you're coordinating work between two or more classes and there's no polymorphism to take advantage of. If you can't decide what class a method belongs to, the answer is often *neither*.

Finally make sure to learn about `@singledispatch`; it will feel like magic if you haven't yet.

As a bonus, following these guidelines gives you objects with *excellent testability*.

Beyond the Snake's Nose

The last approach is so useful that it sneaked into we-don't-subclass Go under the moniker of *embedding*:

```
type EmailAddr struct {
	addr string
}

type Mailbox struct {
	EmailAddr
	pwd string
}
```

Instances of `Mailbox` now have an attribute `addr` as if it was defined within it: https://play.golang.org/p/WSjJA6MYUDb. But you still have to be explicit when initializing and there's no actual *hierarchy*. There's no `super()`. You can only call *side-ways*. A pragmatic compromise!

Looking back, it's the syntax of our approach 3, but in many ways, you get the classes from approach 1.

Seeing this in *Go* was a bit of a revelation to me because my own gut-based subclassing heuristics fit this pattern, but I didn't know how to formulate them. Now I can just say that I use subclassing when I *could* – and *would*! – use embedding in *Go*.

> This goes to stress how worthwhile it is to learn other programming languages and cross-pollinate ideas. However, never forget to inspect the ideas through the unique lens of Python when you try to apply them.

Where to Go From Here

When it comes to reading clarity, properly-done composition is superior to inheritance. Since **code is much more often read than written**, avoid subclassing in general, but *especially* don't mix the various types of inheritance, and don't use subclassing for code sharing. Don't forget that more often than not, **a function is all you need**.

It's important to keep in mind, though, that you can't take your inheritance-based design and just stop subclassing. **A composition-based design is *different* from the ground up**, so you probably will have to replace some of your beliefs and techniques.

Albeit not Python-based, the best intro to OOP design that I'm aware of is *99 Bottles of OOP* and you should read it if you haven't yet. It's not only incredibly instructive but also a fun read.

To not be a total cop-out, I will wrap it up with a concrete example.

Case Study

I'll use edited-for-clarity code from the **wonderful** *Architecture Patterns with Python* that I helped reviewing[12] and that is unconditionally worth your time and money[13]. I'm using it here because Harry – who is one of its authors – told me to write a blog post after I complained about it.

* * *

The goal is an implementation of the *repository pattern*: a class that allows you to add and retrieve objects to and from a data store. For reasons that are uninteresting to this blog post it *additionally* must remember all objects that it either added or retrieved on a field called `seen`.

An important design goal is to keep the actual storage *pluggable*, so it can – for instance – use a database like *Postgres* in production and a dictionary in unit tests. But the code for *remembering* the seen objects is the same for all implementations, therefore you want to share the code.

Specialization doesn't work here, because it goes in the *wrong direction*: the tracking repository is a specialization of a "regular" repository. Thus the code we want to share would end up in the *subclass*. That's useless.

Therefore, the book uses my *least* favorite type of code sharing using subclassing: the *template method pattern*. That means that the base class provides an overall control flow and your subclass fills in some details:

1. The user instantiates a *subclass*,
2. then calls methods on the *base class*,
3. which in turn call methods on the *subclass*.

In this case, the methods the subclasses must implement are `_add_product` and `_get_by_sku`:

```python
class AbstractRepository(abc.ABC):
    seen: set[Product]

    def __init__(self) -> None:
        self.seen = set()

    def add_product(self, product: Product) -> None:
        self._add_product(product)
        self.seen.add(product)

    def get_by_sku(self, sku: str) -> Product | None:
        product = self._get_by_sku(sku)
        if product:
            self.seen.add(product)

         return product

    @abc.abstractmethod
    def _add_product(self, product: Product):
        raise NotImplementedError

    @abc.abstractmethod
    def _get_by_sku(self, sku: str) -> Product | None:
        raise NotImplementedError
```

So, each subclass has to define the `_add_product()` and `_get_by_sku()`

methods. The user then calls `AbstractRepository`'s `add_product()` and `get_by_sku()` methods which in turn delegate to the subclass's `_add_product()` and `_get_by_sku()`, while remembering which objects of type `Product` it has seen[14].

The avid reader will notice the original sin of inheritance right away: it mixes the definition of an interface *and* shares code with the subclass. Check back with *Why inheritance never made any sense* (that I've linked in the intro) if you want a refresher on why that is bad.

The more practical problem is that indirections that go to and fro across class hierarchies are tedious to follow when trying to understand program flow.

And that's true even as a user, because the public API is defined by the abstract base class – not the class you've actually instantiated! This is often not well-handled in documentation systems and you have to jump around while reading.

* * *

When faced with code like this that you want to free of the shackles of subclassing, you're left with two options:

1. **Wrap the class**. Instead of making it a part of `self`, store it in an instance attribute. Delegate to the methods on the attribute as needed.

2. **Parametrize behavior**. This is the way to go once you need to customize the behavior of a class across multiple axes and code sharing via subclassing falls apart. It sounds complicated, but Sandi Metz demonstrates it perfectly in the aforementioned *Nothing is Something* talk by making ordering and formatting customizable in few lines of code.

   Grasping *this* concept is usually when it "clicks" for most people – at least it did for me.

Our example is simple: we only want to do what a concrete repository is doing *plus* something else[15]. Therefore, we go with option number one. If you squint a bit, you'll realize that the way template subclassing was done here is nothing but wrapping a class. Except that the namespaces are mixed up and the control flow is confusing.

The Repository

Instead of an abstract base class with a bunch of code, we define the interface that we'll wrap by defining a protocol called `Repository`:

```python
class Repository(typing.Protocol):
    def add_product(self, product: Product) -> None: ...
    def get_by_sku(self, sku: str) -> Product | None: ...
```

Of course, if you don't use type annotations, you can leave this step out.

*   *   *

A simple implementation that uses a dictionary to store the data could look like this:

```python
class DictRepository:
    _storage: dict[str, Product]

    def __init__(self):
        self._storage = {}

    def add_product(self, product: Product) -> None:
        self._storage[product.sku] = product

    def get_by_sku(self, sku: str) -> Product | None:
        return self._storage.get(sku)
```

The repository has to implement the two promised public methods, but the whole class *belongs to it*. There's never any danger of name collisions. It has only one job: saving and retrieving `Product`s. It also doesn't have to know that a protocol called `Repository` even exists; your type checker will figure it out for you that it's an implementation of it.

Tracking

Next, let's implement the tracking on top of `Repository` by wrapping an instance of it:

```
class TrackingRepository:
    _repo: Repository
    seen: set[Product]

    def __init__(self, repo: Repository) -> None:
        self._repo = repo
        self.seen = set()

    def add_product(self, product: Product) -> None:
        self._repo.add_product(product)
        self.seen.add(product)

    def get_by_sku(self, sku: str) -> Product | None:
        product = self._repo.get_by_sku(sku)
        if product:
            self.seen.add(product)

        return product
```

This class is *composed* of an object of which you only know that it implements `Repository`, and a set of `Product`s. If you use anything on the `_repo` attribute that is not promised by the `Repository` interface, your type checker will yell at you without having to execute the code.

Summary

I like this version much better, because it has a crystal clear program flow. You know where methods and attributes are coming from without checking any base class(es).

The price for this clarity is that we have to store the repository on our class (`_repo`) and call `self._repo.add_product()` instead of `self._add_product()`. That's a bit more typing.

On the other hand, we've ended up with two small, independent classes whose only contract is a tight, explicit interface. This is not only easy to **read** and **comprehend**, but also easy to **test**.

As a parting epiphany: If you always wondered how to write tests for code that isn't just string manipulation or adding two numbers like in all testing tutorials, I hope you see now that learning better OOP design will conveniently help you with that too.

Final Words

Thanks for sticking with me through the whole thing! My ultimate goal is to add more nuance to the discussion. I want to make you understand that using `Exceptions` doesn't make it a good idea to also use the template method pattern, because "both are subclassing". I hope I have somewhat succeeded.

Due to its length, this article is unlikely to get a lot of "*see, read, retweet / upvote*" sharing. Most likely, it spent some time in an open tab / your reading queue too! Therefore, it would be great if you could share it in any way to help it spread nevertheless.

Feel free to let me know what you think about it or how many coffees it took you to get through. I don't plan on spending much time on public discussions, because they tend to get too heated, pedantic, and dogmatic. One of the reasons I wrote this article is to be able to just point other participants to its URL and use the ejection seat. May it serve you the same way!

I'm working on a talk based on this material, so get in touch if you'd be interested in a presentation at your conference or company!

* * *

1. I'm explicitly not going to talk about standard library APIs. Yes, `SimpleHTTPServer` requires you to subclass, but that's an API decision and not inherent to Python's design. ↩

2. While very insightful, the linked article may be difficult to grok, because it uses a vernacular that could be foreign to you depending on your experience with other programming languages. You *don't* need to read it to understand this blog post. On the other hand, you may find it easier to understand *after* reading this blog post. ↩

3. Completely unrelated to `typing.Protocol` that we'll talk about later. ↩

4. I'm naming *Twisted* here because I was part of the core team when we realized the errors of our ways. It's an acknowledged error, not shade. ↩

5. I would argue that the class is too big if it comes so far. Another reason why I'm usually only doing it to classes that I don't control. ↩

6. Somewhat confusingly, this is also called "implementing a protocol". Searching the Python documentation for the word *protocol* yields an eclictic collection of results. ↩

7. For instance, it's very common for `Exception`s to only carry the information that they are a subtype of, say, `ValueError` and add no new methods, attributes, or behaviors. ↩

8. The common example is code – that is written for rectangles – assumes being able to manipulate the width and the height *independently*. So an implementation of squares has to either silently change the height whenever the width is changed (and vice versa), or raise an error. ↩

9. The L in SOLID. ↩

10. I'm keeping the example short for brevity. Obviously it wouldn't make sense to make such a big deal for two types with two fields each. ↩

11. Attributes/fields that can be `None`. ↩

12. You can find it by searching for 'Does it "make you want to beat Harry around the head with a plushie snake"?' which were my exact words about the section that discusses it. ↩

13. You can read the book for free on the web. But if you have budget for books, I would love it if you could support Harry and Bob. This book is very important and I want them to be rewarded for writing it. I'm not getting any commission – I just get the fame of having my name on the back cover next to Brandon Rhodes's. 🤩 ↩

14. For our discussion, the shape of `Product` is not interesting except that it has an `sku` field of type `str`. ↩

15. Technically, this is the original *Decorator Pattern*. We maintain the API and add behavior. ↩