Why Change?

Life is Better Without It

Moshe Zadka – https://cobordism.com

2020

Shared Mutable State is Bad

Options:

- Don't share
- Don't mutate

Shared mutable state is bad. Sometimes people associate that badness with threads, but threads are just one example of how shared mutable state is bad. In Python, it's not even *particularly* bad. Your dictionary will still stay a dictionary. You just might find that the value of a key is different than you expected it tor be. But, after all, this can happen even if you sent this dictionary to a function. Or made the dictionary be a class or instance variable. Or had a function capture it via a closure, then send it to another object, which finally mutated it.

If shared mutable state is bad, there are only two ways to avoid it. One way is to avoid sharing. This is *really* hard in Python. Objects are thrown around all the time. The only other way is to avoid mutating.

Avoid Sharing?

What about

- Modules
- Function defaults
- Class variables
- Arguments

But, after all, this can happen even if you sent this dictionary to a function. Or made the dictionary be a class or instance variable. Or had a function capture it via a closure, then send it to another object, which finally mutated it.

Avoid Mutating!

Much better.

If shared mutable state is bad, there are only two ways to avoid it. One way is to avoid sharing. This is *really* hard in Python. Objects are thrown around all the time. The only other way is to avoid mutating. In this talk, we will explore programs without change. Not only will this neatly avoid shared mutable change, I will explore other benefits from having constancy.

Digression: Are Squares Rectangles?

...and why would anyone care?

But first, I want to take you on a little digressions: are squares rectangles? Well, a rectangle is a shape with 4 sides and 4 right angles, and a square is a shape with 4 sides and 4 right angles where all the sides are the same. So it sound like every square is a rectangle.

We are not in a math conference, though. We came here to talk about Python. How would we implement a rectangle in Python? We would write an Interface, of course. (You might write ABCs, or Protocols: they have the same problems, but they are easier to see with some of the tooling around interfaces.)

What's a Rectangle (in Python)

```
class IRectangle(Interface):
    def get_height() -> float:
        """Return height"""
    def get_width() -> float:
        """Return width"""
    def set_height(height: float):
        """Set height"""
    def set_width(width: float):
        """Set width"""
```

This is the most straightforward interface I can think of. We want to be able to get the height and the width, and we want to be able to set them. With this interface written, it is time to start implementing it. So let's write a square class that implements this interface.

What's a Square (in Python)

```
@implementer(IRectangle)
@attr.s(auto_attribs=True)
class Square:
    _side: float
    def get_height(self) -> float: return self._side
    def get_width(self) -> float: return self._side
try: verifyClass(IRectangle, Square)
except Exception as exc:
    print(textwrap.dedent(str(exc).split(":")[1]
          ).strip().replace("__main__.", ""),
          file=sys.stderr)
```

```
The IRectangle.with_height(height) attribute was not provided
The IRectangle.with_width(width) attribute was not provided
```

We were in such a rush to write the class we forgot to implement a couple of methods. This happens even to the best of us. Different ways of specifying interfaces in different ways. With ABC, this would make Square an abstract class that cannot be instantiated, so it would probably fail when you unit test. With Protocol, it would fail when you run mypy. But however you do it, this is not a problem – we'll just dive right in and implement those!

What's a Square (in Python) (Fixed)

An easy mistake – we forgot a couple of methods.

Let's fix that.

```
@implementer(IRectangle)
@attr.s(auto_attribs=True)
class Square:
    _side: float
    def get_height(self) -> float: return self._side
    def get_width(self) -> float: return self._side
    def set_height(self, height: float):
        self._side = height # ???
    def set_width(self, width: float):
        self._side = width # ???
```

But there is something weird when we set the length of the sides. The two methods have the same implementation! Not just that, but modifying the height modifies the width. But is this so bad?

Well, what do people do with rectangles?

What Do You Do With a Shape (in Python)

```
def area(rectangle: IRectangle) -> float:
    return (rectangle.get_height() *
            rectangle.get_width())
def double_height(rectangle: IRectangle) -> float:
    rectangle.set_height(
        2 * rectangle.get_height())
```

The goal of writing an interface is to be able to have code that can work on any object that conforms to the interface. One nice things about rectangles is that we can write a function that will calculate the area of a rectangle given our interface: we just multiply the width and the height. We can also write a function that will grab the height, and set the height to double that. This is a fun way to stretch a rectangle.

What Do You Do With a Shape (in Python) (Cont.)

```
x = Square(side=5)
print(area(x))
double_height(x)
print(area(x))
```

```
25
100
```

But what happens when the generic code meets a square? You tried to stretch the rectangle, but instead you quadrupled, not doubled, the area. Because the square had to make sure all sides are the same, you couldn't stretch it. In some sense, the interface we specified was not compatible with squares. So, are squares not rectangles? Is math dead?

Let's Stop Mutating

```
class IRectangle(Interface):
    def get_height() -> float:
        """Return height"""
    def get_width() -> float:
        """Return width"""
    def with_height(height: float) -> IRectangle:
        """Rectangle with same width, new height"""
    def with_width(width: float) -> IRectangle:
        """Rectangle with same height, new width"""
```

Our enemy was none other than that constant thorn in our side: mutation. Even when there was no sharing, it *still* managed to break our code. As always the solution is just to avoid mutation. Our new interface does not require shapes to change (what a weird notion!) Instead, a rectangle can return a new rectangle with a different height and width. Does that solve our problem?

The Immutable Rectangle

```
@implementer(IRectangle)
@attr.s(auto_attribs=True, frozen=True)
class Rectangle:
    _height: float
    _width: float
    def get_height(self) -> float:
        return self._height
    def get_width(self) -> float:
        return self._width
    def with_height(self, height) -> float:
        return attr.evolve(self, height=height)
    def with_width(self, width) -> float:
        return attr.evolve(self, width=width)
```

Ah, you say, that sounds like a big pain. Instead of changing some attributes, now I need to create a whole new object? Luckily, the attrs library has our back with attr.evolve that makes it no harder than just setting an attribute.

The Immutable Square

```
@implementer(IRectangle)
@attr.s(auto_attribs=True)
class Square:
    _side: float
    def get_height(self) -> float:
        return self._side
    def get_width(self) -> float:
        return self._side
    def with_height(self, height: float) -> IRectangle:
        return Rectangle(width=self._side,
                         height=height)
    def with_width(self, width: float) -> IRectangle:
        return Rectangle(height=self._side,
                         width=width)
verifyClass(IRectangle, Square)
```

```
True
```

But we could already have a "canonical" implementation of a rectangle. The problem we struggled with was implementing an IRectangle-compatible *Square*. Can we do that? We did try to do it before, but forgot a couple of methods. Now we made sure we verified the interface, but we ended up being technically compatible last time too. What changed?

What Do You Do With an Immutable Shape (in Python)

```
def double_height(rectangle):
    return rectangle.with_height(
        2 * rectangle.get_height())
x = Square(side=5)
print(area(x))
print(area(double_height(x)))
```

```
25
50
```

Because Square.with_height did not have to return a square, it could return the right thing: a rectangle that only has its height stretched. Immutability fixed our code. Now the area of the stretched rectangle is 50, as we expected.

Let's Get Back to Sharing

At some point, someone told you not to do this. Do you remember why?

```
def sum_with_extra(e1, e2, things=[]):
    things.append(e1)
    things.append(e2)
    return sum(things)
```

Modern linters would warn on this code, even though it is completely legal. Maybe you didn't learn about this from a linter – maybe a friend, a colleague, or a teacher explained that this is a bad idea. Hopefully, you have not done this since, so you might not remember quite why it was a bad idea.

Let's try and remember.

A Bad Trip Down Memory Lane

```
sum_with_extra(1, 2, [3, 4])
```

```
10
```

```
sum_with_extra(1, 2)
```

```
3
```

```
# Whoops!
sum_with_extra(1, 2)
```

```
6
```

Ah, yes. Because the default argument is only computed at function definition time, now it is *shared*. Lists, in Python, are *mutable*. Shared and mutable? Sounds like we have a problem on our hands. Indeed, we do! sum_with_extra has "memory". This is not great.

The Fix is Easy!

```
def sum_with_extra_v2(e1, e2, things=None):
    if things is None:
        things = []
    things.append(e1)
    things.append(e2)
    return sum(things)
```

This is why there is a standard idiom to fix this. It does add two lines of code to our three line function, almost doubling it in size, but you have probably written this code so many times you barely noticed. This is more of a "boilerplate" and less of an "idiom". But, we all do it, and then our code is correct.

Everything is Awesome!

```
sum_with_extra_v2(1, 2, [3, 4])
```

```
10
```

```
sum_with_extra_v2(1, 2)
```

```
3
```

```
sum_with_extra_v2(1, 2)
```

```
3
```

```
things = [1, 2]; sum_with_extra_v2(1, 2, things)
```

```
6
```

```
# Whoops!
sum_with_extra_v2(1, 2, things)
```

```
9
```

OK, maybe the word "correct" was a bit premature. Seems like our code still has some kinks to work out. If we modify the argument, and someone still holds a reference, then we are going to have a pretty bad time. Their object will change, without them knowing. Once again, we shared a mutable object, and paid the price.

One Urgent Hot Fix Later...

We got it to work!

```
def sum_with_extra_v3(e1, e2, things=None):
    if things is None:
        things = []
    things = things.copy()
    things.append(e1)
    things.append(e2)
    return sum(things)
```

OK, so technically we can go back to things=[] but we already have our linter set to automatically shock anyone who commits this thing after our last incident. So all we had to do was add three lines of boilerplate and we are all good. Three versions, double the code, and our problems are gone. Guess mutability is not so bad after all, huh?

..Meanwhile, Without Mutation

Let's throw caution to the wind and live our best life.

```
def sum_with_extra_p_v1(e1, e2, things=v()):
    things = things.append(e1)
    things = things.append(e2)
    return sum(things)
```

But what if we had chosen the correct fork in the road, and went with immutable objects. We can use the pyrsistent library and have fun immutable sequences. Now we can have our cake and eat it. Our code is almost the same as the one in our original v1. The first version forgot the things = in the beginning, but this did not work for any inputs, so we fixed it immediately.

We Don't Need v2

```
sum_with_extra_p_v1(1, 2)
```

```
3
```

```
sum_with_extra_p_v1(1, 2)
```

```
3
```

```
things = v(1, 2)
sum_with_extra_p_v1(1, 2, things)
```

```
6
```

```
sum_with_extra_p_v1(1, 2, things)
```

```
6
```

There is no version 2. Version 1 did not have the first problem, and it didn't have the second problem. It just... worked. With the most obvious code. With no boiler plate. Because when we throw away mutability, we get good things.

But Nested Data Structures Are a Drag?

How do you increase the hits on web_1?

```
stats = m(
    frontend=m(
        web_1=m(hits=53),
        web_2=m(hits=78)),
    backend=m(
        db1=m(queries=23),
        db2=m(queries=11)))

# This doesn't work:
# stats["frontend"]["web_1"]["hits"] += 1
```

But nested immutable datastructures, right? They are a pain. Where with a dict-of-dict we could have just done the obvious += 1, now we have to get a new inner-inner-dictionary, then update so we can have an inner-dictionary, and then, finally, we update the top-level dictionary. This sounds like a lot of boilerplate again. We just moved the boilerplate around!

Like This

```
new_stats = stats.transform(
    v("frontend", "web_1", "hits"),
    lambda x: x + 1)
pprint.pprint(pyrsistent.thaw(new_stats),
              width=50)
```

```
{'backend': {'db1': {'queries': 23},
             'db2': {'queries': 11}},
 'frontend': {'web_1': {'hits': 54},
              'web_2': {'hits': 78}}}
```

Well, maybe there is a better way. Pyrsistent supports deep “transforms” that do all of those things for you, and just return a new top-level dictionary. This is a little bit more code than with regular nested dictionaries, but not much more. The biggest thing is having to define an anonymous function to specify the “mutation”.

Conclusion

- Sharing good
- Mutation bad
- Share more
- Mutate less
- Be happy