Boring Object Orientation

Boring is better than interesting

Moshe Zadka – https://cobordism.com

2019

Python and object oriented programming

Everything is an object

In Python, everything is an object. Python is based, roughly, on the Smalltalk model of objects. It is inspired by the tradition of object-oriented programming.

Why OO design principles?

Guidelines to code that is easy to maintain

As object-oriented programming matured, the discipline of *object-oriented design* started forming. This discipline is meant to give guidance about mapping the *problem domain* to the *class and object structure*.

Concepts like the difference between has-a and is-a belong in this discipline. Another famous concept in the discipline is SOLID: Single responsibility principle, Open-closed principle, Liskov substitution, Interface segregation, and Dependency inversion.

Do OO design principles work?

Yes ...but

In general, good OO princples do apply to Python. They have been carefully thought out, and are worthwhile to consider. However, the *way* the apply to Python is sometimes not obvious. Often, taking the concepts too literally means using Python badly.

Principles

Make your objects more boring! The simple tricks that they don't want you to know!

- Declare interfaces
- Simplify initialization
- Avoid mutation
- Avoid hiding
- Avoid methods
- Avoid inheritance

Why declare interfaces?

Explicit is better than implicit

A central concept in OO programming is that of the "interface". Traditionally, people have written in "duck-type Python", which is a 90s name for what would be called, in the 21st century, "if it fits, it sits". The way to know if an object will fit a method is to send it, and as long as it responds correctly to attribute access, everything is fine.

This led to a lot of gnashing of teeth around questions like "is every sequence slicable"? and "what is an easy way to say 'file object that can't be seeked". In

a complex system, it is useful to be explicit about the interfaces intended to be implemented.

Declaring interfaces with zope.interface

```
from zope import interface

class ISprite(interface.Interface):

    bounding_box = interface.Attribute(
        "The bounding box"
    )

    def intersects(box):
        "Does this intersect with a box"
```

One of the ways we can explicitly declare interfaces is with zope.interface. This library focuses on interfaces: declaring them, declaring who implements them and verifying that implementors adhered to the contract. Interfaces are *not* usable as superclasses.

Also note that the *interface* methods are *not* declared with self. The reasoning is that the interface shows how the method will be called. Because they are not super-classes, interfaces can be used to define data attributes. This is important for a language like Python, where part of the public contract might well be "you can access x to get the x co-ordinate".

Testing for interface provision

```
from zope.interface import verify

def test_implementation():
    sprite = make_sprite()
    verify.verifyObject(ISprit, sprite)
```

It is possible to test that implementations comply with interfaces. Of course, the test is often partial: it will not test anything only mentioned in the documentation, and will not even test that the methods can be called without exceptions! However, it does check that the right methods and attributes exist. This is a nice addition to the unit-test suite, and at list will prevent simple misspellings from passing the tests.

Interesting constructor

```
class Stuff:

    def __init__(self, fname):
        # Create a new object
        self.destination = Destination()
```

```
        # Call a system call
        self.finput = open(fname)
```

Interesting constructors might create a new object, or engage in IO methods. This means that the object cannot be created without it. It ties the object inexorably to the specifics of the IO and to the specific class it creates. This is often a premature binding. Doing IO means having to work around the need for IO in some cases, such as unit tests. It also means we have a defined a specific operating-system resource, in this case a file, while nowhere else in the code do we need to make that assumption.

Finally, when constructors *fail* we are left, temporarily at least with an object, self, which is invalid. This means that it might be impossible to print it in a debugger or a verbose stack trace since the "repr" implementation might refer to the "finput" attribute.

Boring constructor

```
class Stuff:

    def __init__(self, finput, destination):
        self.destination = destination
        self.finput = finput

    @classmethod
    def from_name(cls, name):
        # Create a new object
        destination = Destination()
        # Call a system call
        finput = open(fname)
        return cls(finput, destination)
```

In contrast, boring constructors don't do anything. They just copy arguments to object attributes. This means they cannot fail, and we can easily substitute other input variables if we need different behaviors. This means unit testing is made simpler, and we have a well defined object at any time.

The original logic was important though. The best place for it is often in a so-called "named constructor": a class method which builds the requisite underlying parameters, and then creates the object. The nice thing is that named constructors *can* fail. When they fail, they have no partially constructed object.

Not only can they fail, but they can do more interesting things. For example, named constructors can be async methods which do not return the object but rather a future of an object that can be "await"ed upon.

Why boring constructors

- No partial objects

- Easier testing

Boring constructors make our code more boring. More predictable. It is easy to guess what is a complicated operation and what is not. Because of that, we can read and understand code better. We can make code that is easier to test, and even easier to write logs from.

Using attrs

```
import attr

@attr.s(auto_attribs=True)
class Stuff:
    finput: Any
    destination: Any
```

The "downside" of boring objects is that they are boring to write. The constructor ends up being duplicated code: every single attribute name is written three times. The temptation to spice things up a bit is irresistable.

An alternative to spicing things up is to avoid the temptation altogether. By using attrs, the computer writes the boring constructor for us. Doing boring things is one things which computers absolutely love! By making our constructor boring, we got to the point where a dumb computer can write it.

This frees us up to do the fun parts!

Immutable objects

```
>>> @attr.s(auto_attribs=True, frozen=True)
... class Stuff:
...     destination: Any
...     finput: Any
...
>>> my_stuff = Stuff(Destination(), io.StringIO())
>>> my_stuff.finput = io.StringIO()
Traceback (most recent call last):
...
    raise FrozenInstanceError()
attr.exceptions.FrozenInstanceError
```

Once we have decided to use attrs to write boring objects, we can take advantage of another feature it offers for free: immutability. Immutable objects solve the age-old dilemma of avoiding shared mutable state. Solving it with careful sharing is hard: why not attack the other horn of the dilemma, and cut off the immutability? Immutability is a powerful guarantee.

Immutablity as bug avoidance

```
def some_function(some_list=[]):
    pass
```

We can use immutability to avoid bugs. For example, instead of accidentally sharing a mutable object as an implicit global, and get weird action at a distance, we can... just not! We share the object, but now that it's immutable, we do not have to worry who else has a reference.

Immutablity as interface simplifying

No variation, no invariant breakage!

Immutability often lets us make more data attributes public. We do not need to be concerned about them changing, so we can allow direct access. This is one powerful way to avoid either Java-style getters or Python's distance cousin, the properties.

Frozen attrs

```
>>> @attr.s(auto_attribs=True, frozen=True)
... class Point:
...     x: float
...     y: float
...
>>> origin = Point(0, 0)
>>> up = attr.evolve(origin, y=1)
>>> origin, up
(Point(x=0, y=0), Point(x=0, y=1))
```

Immutabitliy means we cannot change objects in-place. However, with boring objects and attrs' evolve functionality, creating a copy which only modifies a handful of attributes is useful. This means that we can easily do counters or other "mutable" objects by creating near copies. Python will take care of deleting objects with no references, but no shared references will be modified!

Private methods

```
class HTTPSession:
    def _request(self, method, url):
        pass
    def get(self, url):
        return self._request('GET', url)
    def head(self, url):
        return self._request('HEAD', url)
```

Private methods are problematic. Why are they private? Presumably, it is possible to misuse them. How carefully do we document what is correct use? Very often, not at all. They are usually the result of semi-mechanical refactoring. Instead, it is better to make them *public methods* on a *private instance variable*. This will allow them to be clearly documented and tested.

The class can still be in an internal module. But now the class will have clear documentation and contract.

Refactoring private methods away

```
class RawHTTPSession:
    def request(self, method, url):
        pass
class HTTPSession:
    _raw: RawHTTPSession
    def get(self, url):
        return self._raw.request('GET', url)
    def head(self, url):
        return self._raw.request('HEAD', url)
```

The process of removing private methods is straight-forward: check which instance variables they touch, move those to a separate class, and then move the method. It is often the case that the set of instance variables will make sense as its own class.

Methods

```
@attr.s(auto_attribs=True, frozen=True)
class Point2D:
    x: float
    y: float

    def distance_from_origin(self):
        return (self.x**2 + self.y**2) ** 0.5
```

Methods

```
@attr.s(auto_attribs=True, frozen=True)
class Point3D:
    x: float
    y: float
    z: float

    def distance_from_origin(self):
        return (self.x**2 +
                self.y**2 +
                self.z**2) ** 0.5
```

Finally, we need to ask the ultimate question: are methods even useful? Sometimes they are! I am happy Python has methods. But maybe, just maybe, we can avoid using them for everything. Why not write functions? Since all the attributes above are public, we could implement a function.

The usual answer is "polymorphism". The right logic for distance_from_origin regardless of the class. The caller does not have to care! This is what methods accomplish.

Why not methods?

Bloats classes

In return, methods can make classes big and cumbersome. Distance from origin? What about "double"? Or a general "increase by factor of X"? What about the New York norm (sum of all absolute values)? The max norm? The p-norm? Sure, I'm a math geek, but these are points, math objects.

If they were web requests, all the web geeks would be thinking up methods!

singledispatch example

```
@attr.s(auto_attribs=True, frozen=True)
class Point2D:
    x: float;
    y: float

@attr.s(auto_attribs=True, frozen=True)
class Point3D:
    x: float
    y: float
    z: float
```

The core parts of the class fit on one slide. There is no logic to obscure what points are. The logic, helpfully, fits on a different slide which captures just that.

singledispatch example

```
@functools.singledispatch
def distance_from_origin(pt):
    raise NotImplementedError(point)

@distance_from_origin.register(Point2D)
def distance_2d(pt):
    return (pt.x**2 + pt.y**2) ** 0.5

@distance_from_origin.register(Point3D)
def distance_3d(pt):
    return (pt.x**2 + pt.y**2 + pt.z**2) ** 0.5
```

We can have polymorphism without methods. Singledispatch solves that problem neatly. We can put all the functions together, and have the right code switch. If someone adds their own class, they can register their own implementation. Code organization is divorced from the needs of polymorphism.

Inheritance-as-API: Examples in the wild

```
# From the Twisted tutorial
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.write(b"No such user\r\n")
        self.transport.loseConnection()
```

Inheritance-as-API: Examples in the wild

```
# From the Django tutorial
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]
```

Inheritance-as-API: Examples in the wild

```
# From the Jupyter documentation

class EchoKernel(Kernel):
    implementation = 'Echo'
    implementation_version = '1.0'
    language = 'no-op'
    language_version = '0.1'
    language_info = {
        'name': 'Any text',
        'mimetype': 'text/plain',
        'file_extension': '.txt',
    }
    banner = "Echo kernel - as useful as a parrot"

    def do_execute(self, code, silent, store_history=True, user_expressions=None
                   allow_stdin=False):
        if not silent:
            stream_content = {'name': 'stdout', 'text': code}
            self.send_response(self.iopub_socket, 'stream', stream_content)

        return {'status': 'ok',
                # The base class increments the execution count
                'execution_count': self.execution_count,
                'payload': [],
                'user_expressions': {},
```

}

A lot of frameworks and systems have as their official API "take this class and subclass it". One of the earliest to take this tack was Twisted. The way to write a protocol parser is to inherit from the "Protocol" class. But we were not the only ones! Django quickly followed in our footsteps. The official tutorial first few code examples cover how to make a new model: by inheriting from the "Model" class. Not to be left behind, newcomer Jupyter, winner an ACM award, suggests writing a new kernel by... inheriting from the "Kernel" class.

If everyone is doing it, does that mean it is a good idea?

Inheritance-as-API: Issues

"Shared everything"

A subclass shares "everything" with its parent class. This means even private methods in the superclass can interact in funny (and surprising) ways. We do not like private methods, but even private attributes pose similar issues.

Composition

- Define *interface*
- Useful behavior in *referred class*

In contrast, composition allows to separate the two concerns: what interface are we supposed to be *exporting*? This is a matter of documentation. The other concern is "what can help us achieve that interface": a class that supplies some interface: potentially a sub-interface, but not necessarily.

For example, the exported interface might be something like "key value store", and the helper class might just supply raw IO primitives. That does not matter. The external interface is clearly specified, and the helper class's interface is clearly specified.

Everything in its place, and a place for everything.

Composition: Simple example

```
class IMovable(interface.Interface):
    x_position = interface.Attribute()
    y_position = interface.Attribute()
    def tick():
        pass
```

Composition: Simple example

```
@interface.implementer(IMovable)
@attr.s(auto_attribs=True):
class StraightLine:
    dx: float
```

```
    dy: float
    x_position: float
    y_position: float
    def tick(self):
        self.x_position += dx
        self.y_position += dy
```

Composition: Simple example

```
@interface.implementer(IMovable)
@attr.s(auto_attribs=True):
class Sprite:
    _movable: IMovable
    @property
    def x_position(self):
        return self._movable.x_position
    @property
    def y_position(self):
        return self._movable.y_position
    def tick(self):
        return self._movable.tick()
```

The frequent complaint is that this leads to a lot of "explicit forwarding". This can be mitigated by auto-generating code based on the interface. I would recommend Twisted's proxyForInterface here... except that its official API is, you guessed it, by inheritance. Nonetheless, it is a good piece of inspiration.

Python: Language of the free

Diamond inheritance with overriddable constructors as mandatory interface? Sure!

Python will let you get away with a lot of things. You can use complicated super-based hierarchies to manage your API. You can vary the arguments to dunder-init. Nothing will stop you.

With Great Power

Diamond inheritance with overriddable constructors as mandatory interface? ....Maybe not!

Except your good sense and your desire for maintainable code. Just knowing the Python language semantics is not enough. Care must be given to writing your code in a way that makes maintaining it easy, and that makes using it pleasant.

Lessons Learned

- Do as we say, not as we do

- Big systems, big headaches

The wrong lesson to take from big Python systems is how they are written. Most of them were written by people who have learned the lessons as they wrote them. This means that often, some warts were left there. Instead of copying them, talk to people who wrote them and ask how to do it.

Less interesting code

Be dumb as possible when writing code.

Maintaining code is hard. Write simpler code to make maintainance easier.