Boring Object Orientation

Boring is better than interesting

Moshe Zadka – https://cobordism.com

PyBay 2019

Python and object oriented programming

Everything is an object

Why OO design principles?

Guidelines to code that is easy to maintain

Do OO design principles work?

Yes

Do OO design principles work?

Yes ...but

Principles

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

- Declare interfaces

Principles

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

- Declare interfaces
- Simplify initialization

Principles

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

- Declare interfaces
- Simplify initialization
- Avoid mutation

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

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

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"
```

Testing for interface provision

```
from zope.interface import verify

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

Interesting constructor

```
class Stuff:

    def __init__(self, fname):
        # Create a new object
        self.destination = Destination()
        # Call a system call
        self.finput = open(fname)
```

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)
```

Why boring constructors

- No partial objects
- Easier testing

Using attrs

```
import attr

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

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
```

Immutablity as bug avoidance

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

Immutablity as interface simplifying

No variation, no invariant breakage!

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))
```

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)
```

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)
```

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

@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)
```

Why not methods?

Bloats classes

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
```

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
```

Inheritance-as-API: Examples in the wild

- Twisted

Inheritance-as-API: Examples in the wild

- Twisted
- Django

Inheritance-as-API: Examples in the wild

- Twisted
- Django
- Jupyter

Inheritance-as-API: Issues

"Shared everything"

Composition

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

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_p
    @property
    def y_position(self): return self._movable.y_p
    def tick(self): return self._movable.tick()
```

Python: Language of the free

Diamond inheritance with overriddable constructors as mandatory interface? Sure!

With Great Power

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

Lessons Learned

- Do as we say, not as we do
- Big systems, big headaches

Less interesting code

Be dumb as possible when writing code.