

OOP - The good parts...

Object Oriented Programming

We're including a brief intro to OOP.

It's extremely common in the python ecosystem and you can't avoid it. Some familiarity is therefore necessary in order to get things done.

It's also extremely easy to shoot yourself in the foot with it. In general, unless you know what you're doing, we'd recommend avoiding it except when interfacing with a library like PyTorch, where you have to extend classes etc.

This intro is very brief and should just cover the relevant terminology for those who don't know it.


Objects

The main idea in OOP is ... objects.

Objects are data with some associated functionality.

In python, everything is an object.


Objects have properties, which are how you access data on them. For example, here we access the property `bar` on an object of type `Foo`:



```
f = Foo("hello")
print(f.bar)
>>> hello
```

Classes

Classes are how we create objects. A class declaration looks like this:




```
class Foo:  
    def __init__(self, bar):  
        self.bar = bar
```

Think of it as a blueprint for creating objects. This is the blueprint for creating a 'Foo' like we saw in the previous slide.

Methods

Classes also have methods. Methods are basically functions that are associated with an object. When you call them, they have access to everything defined on 'self'. E.g.



```
class Foo:
    def __init__(self, bar):
        self.bar = bar
    def baz(self):
        print(self.bar)
f = Foo("hello")
f.baz()
>>> hello
```

Inheritance

Inheritance allows you to 'extend' a class.

My strong advice is to **never** use this construct in your own code.

It is very rare that it is actually the right design choice and it is extremely easy to code yourself into a corner with it.

I introduce it only out of necessity because many ML libraries will force you to extend a class to work with them.

YOU HAVE BEEN WARNED!!

```
class Person:
    def __init__(self, name):
        self.name = name

class Employee(Person):
    def __init__(self, name, salary):
        super().__init__(name)
        self.salary = salary

e = Employee("tom", 10)
print(e.name)
>>> tom
print(e.salary)
>>> 10
```

An example of inheritance in PyTorch

Here's an example of where you might be required to extend a class in PyTorch.

PyTorch provide the 'Module' base class and in order to create a custom model you must extend it.

```
class Net(nn.Module):  
    def __init__(self):  
        super(Net, self).__init__()  
        self.conv1 = nn.Conv2d(1, 32, 3, 1)  
        self.conv2 = nn.Conv2d(32, 64, 3, 1)  
  
    def forward(self, x):  
        x = self.conv1(x)  
        x = F.relu(x)  
        x = self.conv2(x)  
        x = F.relu(x)  
        return F.log_softmax(x, dim=1)
```

If not OOP, then what?

The vast majority of ML work is simple data transformation.

We load data, transform it, and put it into models.

This workload is extremely well suited to a function oriented design. You covered a lot of this stuff in a previous module, so I'd recommend defaulting to that.

An exception to the rule... dataclasses

There is one OOP construct in Python that I do strongly recommend everyone use. That is dataclasses.

Dataclasses provide named records for python - i.e. data with properties that you can reference by name.

They are akin to structs in other languages and I strongly prefer them to passing around dictionaries or tuples of data.

A dataclass example

```
from dataclasses import dataclass
```

```
@dataclass
```

```
class Params:
```

```
    a: int
```

```
    b: str
```

```
    c: float
```

```
p = Params(10, "hello", 1.0)
```

```
print(p.a)
```

```
>>> 10
```

Mastering your tools is an important part of writing software. And I'd encourage you to all explore Python a bit. Two great resources are:

<https://docs.python.org/3/>

<https://docs.python-guide.org/>

Learning to read and use proper documentation is a key part of improving as an engineer.