Classes

What we will cover...

1. State
2. Object-oriented programming
3. Classes
4. Methods and Attributes
5. Loose Coupling

Understanding State

State is an important term in programming.

You can think of state as "state of the world."

Understanding State

Consider your computer right now. The state consists of:

1. Which programs are open
2. Where is your mouse
3. What wifi are your connected to.

Most of what computer programs do is take some state and change it to another. If you couldn't change your programs or move your mouse, you wouldn't use your computer.

Keeping track of state

Consider a typical machine learning problem. Your state might consist of:

1. Data. Often we put our data through a "pipeline", which transforms it from one state to another. At any given time, your data might be in any number of states between "fresh and useless" and "just-the-way-you-want-it."
2. Models. You might have one, or many. They might have the same, or different hyperparameters. They might be trained or not-yet-trained.

What is procedural programming?

When people begin programming, it's natural to think procedurally, to tell the computer what to do:

1. Do one thing
2. Do the next thing
3. Do the third thing

Procedural Programming

This works for simple applications. But it does not scale.

It requires us to understand, in our heads, every step the computer should do.

That means we need to understand, in our heads, the entire state of the system.

To do things more complicated than we can keep in our heads at once, we need to break things down into **component parts**.

Functional programming

We saw that we could use functions as the **component parts** of a complex system.

This can work great, and indeed, forms the core of a paradigm called **functional programming**.

In functional programming, the state of a system is kept in (immutable) data structures and (pure) functions operate on the data to return new versions of the state.

Object oriented programming

Another paradigm for breaking a system into component parts is **object oriented programming**. In this paradigm, we model the world as a set of **objects**.

Those objects keep their own personal state, safe and separate from the rest of the objects in the world.

Their personal state is read and updated via **methods**. Methods are functions that have special access to the secret state of the objects.

Classes

In object-oriented programming, we represent the state of our system in a bunch of objects.

Those objects consist of potentially many **instances** of a few different **classes**.

Let's try and understand this new vocabulary with some examples.

Classes

In a video game, the state of the system might consist of the score(s) of each individual user.

A natural way to model this system is to create "user" objects that each keep track of the state of a single user. To do this:

1. We create a "User" **class** to describe how a user object should function.
2. For each user in our game, we create an **instance** of that class.

Classes

In a data anlysis pipeline, one piece of state that needs to be kept track of is the data!

The data might live in files on a harddrive. We need to keep track of the latest files and where to get them from. To do this:

1. We create a "DataStore" **class** to describe how a datastore object should work.
2. For each type of data we work with, we create an **instance** of the DataStore class.

Class Syntax

This is how we create a class and an instance in python.

Right now, our `DataStore` class does nothing.

But we can already **instantiate** it.

`data_store` is an instance of the `DataStore` class.

```
class DataStore():
    pass

data_store = DataStore()

isinstance(data_store, DataStore)
# True
```

Attributes and Methods

Objects in Python are **instances of a class**.

Methods are special functions that have access to the internal state of objects.

Objects store their internal state in **attributes**.

Let's see how that works

Attributes and Methods

We define **methods** in the class by writing a normal function in the class definition block.

The function must have at least one parameter: `self`.

Methods are called on the instance via dot `.` notation.

`self` is passed automatically to the method.

```
class DataStore():
    def get_data(self):
        # self contains internal state
        return 'foo'

data_store = DataStore()

# NOTE: self is passed implicitly
data_store.get_data()
# 'foo'
```

Attributes and Methods

`self` is shared among all the methods of the object.

`self` contains **attributes**, which are set and retrieved via the `.` notation.

We create methods to set, update, read, or do work with the attributes of an object.

```
class DataStore():
    def set_data(self, data):
        self.data = data

    def get_data(self):
        return self.data

data_store = DataStore()

data_store.set_data('bar')
data_store.get_data()
# 'bar'
```

Attributes and Methods

Python is pretty relaxed language.

Attributes on `self` contains the "internal" state of the object. But Python lets anyone access that internal state directly via attributes on the object itself.

```
class DataStore():
    def set_data(self, data):
        self.data = data

data_store = DataStore()

data_store.set_data('bar')

data_store.data
# 'bar'
```

Constructors

Classes have a special method called `__init__`.

This is called the **constructor**.

The **constructor** is a special that exists to create the "initial state" of the object and set attributes on `self`.

The **constructor** is called when the class is instantiated.

```
class DataStore():
    def __init__(self, data):
        self.data = data

    def get_data(self):
        return self.data

# __init__ called:
data_store = DataStore('bar')

data_store.get_data()
# 'bar'
```

Using classes

Let's create a real `DataStore` class. Each **instance** of `DataStore` is used to access data from a different "location".

Other parts of the program can now use the `data_store` instance without having to know anything about how it gets the data or where.

This means the storage of the data is **decoupled** from the rest of the program.

```
class DataStore():
    def __init__(self, location):
        self.data = location

    def get_data(self):
        # read from self.location
        # get latest data as "data"
        return data

# __init__ called:
data_store = DataStore('path/to/data')

data_store.get_data()
```

Loose Coupling

If with think of our system as a set of component parts, we can begin to think of what would happen if we needed to change one of the parts.

We say that two parts are **tightly coupled** if, when we change something in one part, we need make (big) changes in the other.

We say that two parts are **loosely coupled** if we can change the inner workings of one part without needed to change anything about the other.

Loose Coupling

We can imagine our `data_store` instance being used like this.

How does use of the `DataStore` class enable **loose coupling**?

```
def clean_data(data_store):
    data = data_store.get_data()
    # clean data
    return data

def train_model(data, model):
    # do some training
    return model

data_store = DataStore('/path/to/data')
data = clean_data(data_store)
train_model(data, model)
```

Review

1. State
2. Object-oriented programming
3. Classes
4. Methods and Attributes
5. Loose Coupling