Barcelona GSE Data Science Center

Object Oriented Programming with Python

September 2018

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.

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

To break things down, we need a design pattern to know how to break things down!

Understanding State

To understand object-oriented programming, it's important first to understand the term:

State

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

Barcelona GSE Data Science Center

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 we do when we create computer programs is take some state and change it to another.

In other words, if you couldn't change your programs or move your mouse, you couldn't use your computer.

Keeping track of state

The question then becomes, how do we keep track of this 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 (fitted or not-yet-fitted).
3. Output.

Objects

Object oriented programming seeks to split the state of the world into individual “objects,” which are both responsible for keeping track of their own state, and also responsible for knowing how to change it.

O-O reflects nature:

Consider the forest, with all its animals. No one individual keeps track of each fox in the forest: how much fur they have, how much they have eaten, how thirsty they are, etc.

Each fox is in charge of itself. Nobody can put food in the foxes belly.

Objects

The other feature of the foxes of the forest: they are all alike in their technical inner workings (they have the same type of stomach, same type of mouth). But they might be in a different state at any given moment (one might be hungry, one might be full).

In O-O programming, we reflect this pattern via “classes” and “instances”.

“Fox” is a class.

Each fox in the forest, is an “instance” of the “Fox” class.

Polymorphism

Object oriented programming becomes really powerful via polymorphism.

Again, consider all the animals in the forest. There are many different kind of animals!

But say I want to go through the forest and give them all water.

Maybe I don't need to know the details of how their stomachs work and how the water goes from their mouths to the rest of their bodies. I just need to give water over to them, and let them take care of the rest.

This is single-dispatch polymorphism!

Barcelona GSE Data Science Center

Methods

Again, in O-O programming, ojects are both responsible for keeping track of their own state, and also responsible for knowing how to change it.

“Changing state,” in all the programming we’ve seen, is done via functions.

In O-O programming, we have special functions called “methods”.

Methods are functions that are defined in a class and “attached” to each instance of that class.

Methods will change or interact with the state of the instance in some way.

Models

Before we get into examples, let's bring this back to data science.

In data science, our models are very naturally modelled as "objects."

There are many different types of models. And in a given program, you might have both: many"classes" of models, and many "instances" of each model class.

For example: you might be testing 3 different classifiers, and several different "versions" of each classifier, with different hyperparameters.

But for each model, you want to do the same thing!

Models models models

For each model you need to:

1. Create it
2. Train it
3. Test it
4. (eventually) Use it

Polymorphism in Models

Therefore, we can expect that each model class might have the same methods. For example, a “train” method, where we give it our data (each model should operate on the same data, of course!).

Classifier

How do we create a class? We “construct” it.

How do we construct a class? With a “constructor” method!

In python, the constructor method is called “__init__”:

```
class ForgetfulClassifier():
    def __init__(self, K):
        self.K = K

my_classifier = ForgetfulClassifier(5)
classifier.K
```

Attributes

Again, remember that in O-O programming, all classes are responsible for their own state, and how to modify that state.

They keep track of their "state" in attributes.

"K" is an attribute:

```
class ForgetfulClassifier():
    def __init__(self, K):
        self.K = K

my_classifier = ForgetfulClassifier(5)
classifier.K
```

Classifier

Methods look just like functions. Note, however, they have access to this simple encapsulation of local state: “self”.

“self” refers to the instance itself. This allows each method to interact with, or update, the local state.

For example, when we train a model, or “fit” a model, we might want to change the state, such that the model keeps track of the fact that is has “been fitted”.

Classifier

```
class ForgetfulClassifier():
    def __init__(self, K):
        self.K = K
        self.is_fitted = False

    def fit(self, X, y):
        self.is_fitted = True
```

Barcelona GSE Data Science Center

Classifier

Models work well as objects, because we want each model to keep track of its own “fitted” state, not only the fact that it was fitted, but also the parameters that it learned!

We’ll create a simple classifier that gets trained on some data, then makes a prediction that is average of the last K datapoints it has seen.

Barcelona GSE Data Science Center

Classifier

```
class ForgetfulClassifier():
    def __init__(self, K):
        super(ForgetfulClassifier, self).__init__()
        self.K = K

    def fit(self, X, y):
        self.is_fitted = True
        self.y = y[-self.K:]
        self.prediction = sum(self.y)/self.K

    def predict(self, new_y):
        return self.prediction
```

Barcelona GSE Data Science Center

Inheritance

Another important part of O-O programming is the idea of inheritance and composition.

We will focus on simple inheritance here.

The basic idea of inheritance is to create a “parent” class, which “children” classes resemble. It’s a way to organize our classes into a Tree. Like an org-chart at a company!

Inheritance

Maybe there is a whole set of classifiers that have “predetermined” results. We can create a base class for this type of classifier.

Inheritance

```
class PredeterminedClassifer():
    def __init__(self):
        self.is_fitted
        print('creating a silly classifier')

    def fit(self, prediction):
        self.prediction = prediction
        self.is_fitted = True

    def predict(self, new_y):
        return self.prediction
```

The base version is very simple

Inheritance

Then, our ForgetfulClassifier can inherit from this class:

```
class ForgetfulClassifier(PredeterminedClassifer):
    def __init__(self, K):
        super(KAverageClassifier, self).__init__()
        self.K = K

    def fit(self, X, y):
        self.y = y[-self.K:]
        self.prediction = sum(self.y)/self.K
        self.is_fitted
```

Notes: super, overriding methods, not overriding methods.

Inheritance

This allows us to implement other classes that are similar, and share the code.

This is an important way to achieve DRY code.

Do not Repeat Yourself

Barcelona GSE
Data Science Center