Notebook

March 7, 2019

1 Object-Oriented Programming

Object-oriented programming (OOP) isn't all that special, it's just a particular style of programming that Python is particularly well designed for. This is a short lesson, we won't cover the theory of OOP, or features you might hear about elsewhere like "inheritance"—see your reading in the P4E book for more.

However, we have to say something about OOP in order to enable you to work with many Python libraries, which expect you to be able to instantiate objects and work with them. So here's the short version.

OOP is another kind of mental model for programming. It mostly is useful for organizing the abstractions in your code. One of the key ideas is that groups of related data and functionality can be tied together in the form of objects.

A **class** is an abstract description of a kind of object. A class declaration will say what kind of things an object of that sort can do.

An **instance** is an actual object that you create and make use of in the computer's memory.

Let's look at a trivial example of a class

```
In [1]: class Dog(object):
            def __init__(self, temperament="normal"):
                self.temperament = temperament

            def bark(self, volume="loud"):
                if self.temperament == "vicious":
                    sound = "snarl!!!"
                elif self.temperament == "timid":
                    sound = "wimper?"
                else:
                    sound = "woof."
                if volume == "loud":
                    print(sound.upper())
                else:
                    print(sound)

            def meet_cat(self):
                self.temperament = "timid"
                self.understanding = "inferior"
```

Here's what you need to know about this code.

1. Functions belonging to a class are called **methods**. You can tell that they belong to the class because they are indented under the class declaration in the first line.

2. Every method takes a special parameter to refer to the object on which it is called as the very first parameter. Here, we're using `self` for that, which is the Python convention, but you could use anything else. When you call these methods, you don't supply the value for the self parameter. We'll see how to call them in a moment.

3. Every class has an `__init__` method. The two underscores on either side of the name show you it's a special method (known as "magic methods" in Python slang... there are a number of those, but init is the most important). This is the method that gets called when you **initialize** a class, that is, create an instance of that class.

4. All those references to `self.temperment` are creating (the first time something is assigned to it) and either mutating or examining an **attribute** (also called a property) of a particular instance. Let's look at some examples of instances.

```
In [2]: goodboy = Dog()
        doberman = Dog("vicious")
```

We've created two different instances of `Dog` and each of them has its own set of instance properties. Let's prove that by calling the bark method.

```
In [3]: doberman.bark()
```

```
In [4]: goodboy.bark("quiet")
```

Here are a few things you can observe from those examples:

1. We instantiate a class by calling it as if it were a function. We can pass parameters into its `__init__` method that way too.

2. `doberman` and `goodboy` have different `temperament` properties, hence they make different sounds when we call their `bark()` methods.

3. We call a method on an instance by putting a period after the instance's name and before the method call. We don't pass the instance into the call, but we do pass any other parameters (like `volume` here) that we want.

4. Instances are totally separate. Nothing we do to `goodboy` affects `doberman` or vice versa.

We can also modify existing instance attributes, and we can access them directly with the same dot notation. Let's look at, then modify, doberman's attributes.

```
In [5]: print(doberman.temperament)
```

```
In [6]: doberman.meet_cat()
```

```
In [7]: print(doberman.temperament)
```

```
timid
```

```
In [8]: print(doberman.understanding)

inferior


In [9]: doberman.bark()

WIMPER?


In [10]: doberman.temperament = "vicious"

In [11]: doberman.bark()

SNARL!!!


In [12]: goodboy.bark()

WOOF.
```

As I said, this is the bare minimum to understand what's going on when you see OOP code. This will benefit you in the next lesson, where we'll look at some OOP code to run a simulation. It will also benefit you in interacting with Python libraries which often expect you to know how to instantiate an object, call methods on it, and access its properties.