KENDRA PUGH: Hi. Let's talk about object oriented programming in Python. First, object oriented programming is a programming paradigm. It's in the same category as things like functional programming and imperative programming. But object oriented programming is going to be the programming paradigm that describes most of the code that you're going to interact with in 6.01.

So it's important to understand how it works and how, in particular, you want to be able to code in an object oriented programming paradigm in Python. So today, I'm going to go over a quick crash course on object oriented programming, and also indicate all the little tips and tricks you need in order to program in Python. Let's look at what I have written up.

So the most important thing to remember when you're learning about the object oriented programming paradigm is that everything is an object. And what I mean when I say that, and what people mean when people say that, is that the ideals behind the paradigm are that you interact with your code in the same way that you would interact with objects in the physical world, right?

There's a particular piece of paper on the desk in front of me. And it is a kind of piece of paper. So I know how to interact with it the way you would interact with any other piece of paper.

If you want to codify this in an object oriented programming paradigm, you write up classes. Classes are your basic unit of code block. They describe what a thing can do and what a thing has or what attributes a thing has. And in object oriented programming-- frequently object oriented programming and in object oriented programming in Python-- we refer to those things as methods or functions that a particular object may have, and attributes or particular variables that an object may have.

Once you've codified what any object of a particular class can do, you can then use the code that you've written to instantiate an object. An object is the functional unit

in the object oriented programming paradigm. It's the thing that you interact with, and tell what to do, and produces results for you. It's the thing that makes up-- it and classes those are the two things that you need to think about. But you also have to think about how they're different.

I have a particular sheet of paper in front of me. It has all the properties of a sheet of paper. And when I think about all the things I can do to a sheet of paper, that constitutes a class. But the particular piece of paper that I have is an instance of that class. It's a particular piece of paper.

That's the gist of object oriented programming and the things that you need to know. Now that I've covered them, I'm going to go over the most basic class I could come up with in terms of object oriented programming in Python.

This is a class that specifies what a 6.01 staff member has in terms of an attribute or a method in Python. If I want every instance of a particular class or every staff member of the class, staff member 6.01, to have a particular attribute, I can specify like this. Every instance of Staff6.01 is going to have an attribute room. And that attribute room is going to be set to the string describing 34-501, the 6.01 lab.

If I want every 6.01 staff member to be able to do a particular thing, or have a particular method, or call a particular function, then I specify it like this. This is the beginning of a method in the class Staff6.01. It's called sayHi. I'll talk about self in a second. Don't worry about it. Act as though-- if this is your introduction to Python programming, then pretty much pretend it's not there. It's kind of like this, but we'll cover that in a second.

And if any instance of the Staff6.01 class calls sayHi, then "hello" will be printed to standard out. I have a couple examples up on the board behind me. And if you type them into Idle and see what their return is like, you'll be able to-- after you've typed in this-- you'll be able to interact a little bit better with what Python considers classes, and objects, and that sort of thing.

If you look at type Staff6.01, it'll tell you about a class, which is an object in itself. But

it's a specification for instances of an object. If you want to instantiate an object that is of type Staff6.01, you need to use the parenthesess on the end. This treats Staff6.01 like a call and creates an object. If you just type Staff6.01, you're just reassigning Staff6.01 class to the name kpugh, and that's not useful. Every Staff6.01 member should not be considered a Kendra, right?

Once you have instantiated a particular object of type Staff6.01, you can look at the type of that object, right? Now you've got one object, kpugh, myself. And that is a class of Staff6.01. Likewise, now that you have this object, kpugh, you can look at its attributes and methods. If you look at kpugh.room, then it should print to the screen "34-501." That's because that's the attribute associated with this instance.

If you call kpugh.sayHi(), it will use the method in the class type of this object. So when you call .sayHi(), it looks at kpugh, looks at the type, says, that's of type Staff6.01, goes to class Staff6.01, finds the definition for sayHi(), and then executes this code. Hopefully, that all made sense.

Now, I want to talk about self. And you might say, Kendra, I don't understand where that comes into play. And you didn't even use it over here!

If you're familiar with C++ or Java, self is a lot like this. Self is an implicit argument passed in here. Even though you specify zero arguments, it's considered the first argument. And you'll probably see a lot type errors when you're first programming in object oriented programming that say that you've either passed in too many or too few arguments. It has to do with this definition, with self.

Self says, I am talking about myself. That's not particularly intuitive. But I'll try to explain a little bit more. When kpugh calls .sayHi(), sayHi() always has an implicit reference to whatever called it.

When you look at this code, other instances can call this code, right? If I had an instance of Adam Hartz or an instance of Ike Chuang they would also have access to the method sayHi(). And when they called sayHi(), sayHi() would point back to the class definition, but also have a reference to whatever instance called it.

So when you substitute in this self, you substitute in whatever instance called the method. That doesn't seem particularly useful right now because that class definition does not actually make any use of the self, or the ability to use unique instances of an object as sort of unique storage containers. I'm kpugh. I'm different from Adam Hartz. And therefore, I should be able to have different attributes, or different methods, or things that act slightly differently from the way they do for Adam. I'm going to look at a revised definition of class Staff6.01, and that definition will use self in a way that indicates that you can have different functionality for different instances.

It's right here. Class Staff6.01-- it looks really similar. In fact, the first two lines are exactly the same. We've got a class attribute. That means that every instance of this class is going to have this attribute because it's a class attribute.

If I want different instantiations of my class to have different properties, then I need to explicitly address the initialization of those properties. In Python, when we want to do that, we define the method __init__. __init__ is a very special method. It's got these underscores. I think it's a protected keyword. And it always has the format "self," and then whatever arguments you want to pass in when you're instantiating an object.

__init__ is not exactly a constructor. But for those of you that are familiar with C++ and Java, it acts like a constructor. Immediately after the object is constructed, __init__ is called. And all the set up that is required to set up the object happens. So any time you instantiate an object, all of these things are going to be executed. Or all the things under __init__ is going to be executed.

In particular, we're going to set the attribute greeting to whatever argument we passed in when we instantiated the object. So every instance that we create of this class is going to have the class attribute, room. They're all going to be in the same room. But they're also going to have a greeting. And you have the option of specifying the greeting to be whatever you want.

We're going to make use of this in the method sayHi(), which still only takes the

argument self, or the reference to whatever object called the method in the first place. That reference is going to get substituted in here. So no matter which object calls the method, you will have access to its particular greeting using this syntax. Let's walk through an example.

Let's say I make an example of Adam Hartz, and he is a Staff6.01 member. And his greeting is going to be "hi." Likewise, let's make another instance of me using the new Staff6.01 definition. Make sure you type this in because it's not going to work otherwise. And make my greeting "HELLO," as opposed to just "hi."

If you call the sayHi() method using hartz, then you should get a different result then when you call the sayHi() method using kpugh. But if you call the room method-- or, excuse me. If you were after the room attribute of both instances, then you should get the same result because this is the class attribute definition, whereas this attribute is specific to each instance.

That's all I have to say for now about object oriented programming. In my next video, I'll start to talk about inheritance, which is another really important property in 6.01 and also object oriented programming in Python, and also has some slip ups. So I'd like to talk to you about those next.