PROFESSOR: All right everyone. Let's get started. So today's lecture and Wednesday's lecture, we're going to talk about this thing called object oriented programming. And if you haven't programmed before, I think this is a fairly tough concept to grasp. But hopefully with many, many examples and just by looking at the code available from lectures, you'll hopefully get the hang of it quickly. So let's talk a little bit about objects. And we've seen objects in Python so far. Objects are basically data in Python. So every object that we've seen has a certain type. OK, that we know.

Behind the scenes, though, every object has these two additional things. One is some data representation. So how Python represents the object just behind the scenes and what are different ways that you can interact with the object. So for example, every one of these is a different object. For example, this is the number 1,234. It's a specific object that is of type integer. The number 5 is a different object that's of type integer and so on.

We've seen floats. We've seen strings. We've seen lists. Lists and dictionaries are more complicated objects. Object types. Sorry. But every object has a type, some sort of way that it's represented in Python and some ways that we can interact with them.

OK. So the idea behind object oriented programming is, first of all, everything in Python is an object. We've said that before and in this lecture I think we'll really get at what that means. So we've seen strings, integers, dictionaries, lists. Those are all objects. When we did functions, we saw that we could pass as a parameter another function. So functions were also objects in Python. So literally everything in Python is an object.

So what are the kinds of things we can do with objects? Well, once you have a type, you can create a new object that is of some type. And you can create as many objects as you'd like of that particular type, right? An integer 5 and integer 7. Those all work in a program. Once you've created these new objects, you can manipulate them. So for a list, for example, you can append an item to the end of the list, you can delete an item, remove it, concatenate two

lists together. So that's ways that you can interact with objects.

And the last thing you can do is you can destroy them. So and with lists, we saw explicitly that you can delete elements from a list, or you can just forget about them by reassigning a variable to another value, and then at some point, Python will collect all of these dead objects and reclaim the memory.

So let's continue exploring what objects are. So let's say I have these two separate objects. One is a blue car. One is a pink car. So objects are really data abstractions. So these two cars can be created by the same blueprint. OK? This is a blueprint for a car and if an object is a data abstraction, there's two things that this abstraction is going to capture. The first is some sort of representation. What is going to represent the car, what data represents a car object? And the second is what are ways that we can interact with the object?

So if we think about a car blueprint, some general representation for a car could be the number of wheels it has, the number of doors it has, maybe its length, maybe its height, so this is all part of what data represents the car. OK? The interface for the car is what are ways that you can interact with it. So for example, you could paint a car, right? So you could change its color. You could have the car make a noise and different cars might make different noises. Or you can drive the car, right? So these are all ways that you can interact with the car.

Whereas the representation are what makes up the car. What data abstractions make up the car. Let's bring it a little closer to home by looking at a list. So we have this data type of list, right? We've worked with lists before. The list with elements 1, 2, 3, and 4 is a very specific object that is of type list. Again, we think about it in terms of two things. One is what is the data representation of the list? So behind the scenes how does Python see lists?

And the second is, how do you interact with lists? So what are ways that you can manipulate a list object once it's created? So behind the scenes you have a list, L, which is going to be made up of essentially two things. One is going to be the value at specific index. OK? So at index 0, it has the value 1, right, because it's the first element in the list. And the second thing that represents a list is going to be this second part, which is a pointer. And internally this pointer is going to tell Python where is the memory location in the computer where you can access the element index 1.

So it's just essentially going to be a chain, going from one index to the other. And at the next memory location you have the value at index 1, and then you have another pointer that takes

you to the location in memory where the index 2 is located. And in index 2 you have the value and then the next pointer, and so on and so on. So this is how Python internally represents a list. OK?

How you manipulate lists, we've done this a lot, right? You can index into a list, you can add two lists together, you can get the length, you can append to the end of a list, you can sort a list, reverse a list, and so many other things, right? So these are all ways that you can interact with the list object as soon as you've created it. So notice both of these, the internal representation and how you manipulate lists, you don't actually know internally how these are represented, right? How did whoever wrote the list class decide to implement a sort. We don't know.

You also weren't aware of how these lists were represented internally. And you didn't need to know that. That's the beauty of object oriented programming and having these data abstractions. The representations are private of these objects and they are only known by what you can find out how it's done, but they only should be known by whoever implemented them. You, as someone who uses this class, doesn't really need to know how a list is represented internally in order to be able to use it and to write cool programs with them. OK?

So just find a motivation here before we start writing our own types of objects is the advantages of object oriented programming is really that you're able to bundle this data, bundle some internal representation, and some ways to interact with a program into these packages. And with these packages, you can create objects and all of these objects are going to behave the exact same way. They're going to have the same internal representation and the same way that you can interact with them. And ultimately, this is going to contribute to the decomposition and abstraction ideas that we talked about when we talked about functions. And that means that you're going to be able to write code that's a lot more reusable and a lot easier to read in the future. OK.

So just like when we talked about functions, we're going to sort of separate the code that we talk about today into code where you implement a data type and code where you use an object that you create. OK? So remember when we talked about functions, you were thinking about it in terms of writing a function, so you had to worry about the details of how you implement a function. And then you had to worry about just how to use a function, right? So it's sort of the same idea today. So when you're thinking about implementing your own data type, you do that with this thing called a class. And when you create a class, you're basically going

to figure out what name you want to give your class and you're going to find some attributes. And attributes are going to be the data representation and ways that you can interact with your object.

So you, as the programmer of this class, are going to decide how you want people to interact with the object and what data this object is going to have. So for example, someone wrote code that implements a list class, right, and we don't actually know how that was done. But we can find out. So creating the class is implementing the class and figuring out data representation and ways to interact with the class. Once that's done, you can then use your class. And you use the class by creating new instances of the class.

So when you create a new instance, you essentially create a new object that has the type, the name of your class. And you can create as many objects as you'd like. You can do all the operations that you've defined on the class. So for example, someone wrote the code to implement list class and then you can just use the list class like this. You can create a new list, you can get the length pf the list, you can append to the end of the list, and so on and so on.

So let's start defining our own types, OK? So now you're going to define classes, you're going to write classes which are going to define your own types of objects. So for today's lecture we're going to look at code that's going to be in the context of a coordinate object. And a coordinate object is essentially going to be an object that's going to define a point in an xy plane. So x, y is going to be a coordinate in a 2D plane. So we're going to write code that's going to allow us to define that kind of object.

So the way we do that is we have to define a class. So we have to tell Python, hey, I'm defining my own object type. So you do that with this class key word. So you say class, then you say the name of your type. In this case, we're creating a type called coordinate. Just like we had type list, type string, and so on. This is going to be a type called coordinate. And then in parentheses here, you put what the parents of the class are. For today's lecture, the parent of the classes are going to be this thing called object, and object is the very basic type in Python. It is the most basic type in Python.

And it implements things like being able to assign variables. So really, really basic operations that you can do with objects. So your coordinate is therefore going to be an object in Python. All right. So we've told Python we wanted to define an object. So inside the class definition we're going to put attributes. So what are attributes? Attributes are going to be data and

procedures that belong to the class, OK? Data are going to be the data representations and procedures are going to be ways that we can interact with the object.

The fact that they belong to the class means that the data and the procedures that we write are only going to work with an object of this type. OK. If you try to use any of the data or the procedures with an object of a different type, you're going to get an error because these data and these attributes will belong to this particular class. So the data attributes is, what is the object, right? What is the data that makes up the object? So for our coordinate example, it's going to be the x and y values for coordinate.

We can decide that can be ints, we can decide that we can let them be floats, but it's going to have one value for the x-coordinate and one value for the y-coordinate. So those are data attributes. And procedure attributes are better known as methods. And you can think of a method as a function. Except that it's a function that only works with this particular type of object. So with a coordinate object, in this case. So the methods are going to define how you can interact with the object.

So in a list, for example, we've said that you can append an item to the end of the list, we can sort a list, things like that. So when you're defining methods, you're defining ways that people can interact with your object. So for example, for a coordinate object, we can say that we can take the distance between two coordinate points. OK? And that's going to be a way that you can interact with two coordinate points. And just to be clear, these are going to belong to this class, which means that if you try to use this distance method on two lists, for example, you're going to get an error. Because this distance method was only defined to work with two coordinate type objects.

All right, so let's carry on and continue implementing our class. So we've written this first line so far, class coordinate object. So now let's define attributes. First thing we're going to define are data attributes. Generally you define data attributes inside this init, and this is underscore, underscore, init, underscore, underscore, and it's a special method or function in a class. And the special method tells Python, when you implement the special method, it tells Python when you first create an object of this type, call this method or call this function.

So how do we do that? So let's implement it. So we say df because it's just a function. The name is the special name, init. And we give it some parameters, right, just like any other function. These last two parameters are x and y, which are going to represent how you create

a coordinate object. So you give it a value for the x-coordinate and you give it a value for the y-coordinate.

The self, however, is a little bit trickier. So the self is going to be a parameter when you define this class that represents a particular instance of the class. So we're defining this coordinate object in sort of a general way, right? We don't have a specific instance yet because we haven't created an object yet. But this self is going to be sort of a placeholder for any sort of instance when you create the object. So in the definition of the class, whenever you want to refer to attributes that belong to an instance, you have to use self dot. So this dot notation. And the dot is going to say look for a data attribute x that belongs to this class.

So for methods that belong to the class, the first parameter is always going to be self. It can be named anything you want, but really by convention it's always named self. So try to stick to that. And then any other parameters beyond it are going to be just parameters as you would put in a normal function. OK.

In this particular case, we're going to choose to initialize a coordinate object by two values, one for the x and one for the y. And inside this init method, we're going to have two assignments. The first one says, the x data attribute of a coordinate object. I'm going to assign it to whatever was passed in. And the y data attribute for a particular object is going to be assigned whatever y was passed in.

Questions so far about how to write this init? Yeah, question.

AUDIENCE: [INAUDIBLE]

PROFESSOR: How do you make sure that x and y are inits or floats? So this is something that you could write in the specifications, so the docstring with the triple quotes. So whoever uses the class would then know that if they do something outside the specification, the code might not work as expected. Or you could put in a cert statement inside the definition of the init just to sort of force that. Force that to be true. Great question.

Yeah, question.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Does the x, does this self x and this x have to be the same name. The answer is no. And we're going to see in class exercise that you can have it be different. OK. Great. So this defines the

way that we create an object.

So now we have sort of a nice class. It's very simple, but we can start actually creating coordinate objects. So when you create coordinate objects, you're creating instances of the class. So this line here, C is equal to coordinate 3,4, is going to call the init method. It's going to call the init method with x is equal to 3 and y is equal to 4.

I'm just going to go over here and I wrote this previously, because notice when we're creating an object here, we're only giving it two parameters. But in the init method, we have actually three parameters, right? We have these three parameters here, but when we're creating an object, we only give it two parameters. And that's OK because implicitly, Python is going to say self is going to be this object C, so just by default, OK? So when you're creating a coordinate object, you're passing it all the variables except for self.

So this line here is going to call the init and it's going to do every line inside the init. So it's going to create an x data attribute for C, a y data attribute for C, and it's going to assign 3 and 4 to those respectively. This next line here is origin equals coordinate 0, 0 creates another object. OK? It's another coordinate object whose value for x is 0 and whose value for y is 0. So now we have two coordinate objects.

We can access the data attributes using this dot notation and we've seen that before, right? When we've worked with lists we'd say something like, L dot append, right, when we create a list. So the same dot notation can be used with your own objects in order to access data attributes. So here, this is going to print 3 because the x value for object C is 3, and the next line, print origin x is going to print 0 because the x value for the object origin is 0. OK.

So we've created a coordinate object. We have to find the init method so we have a way to create objects when we use the class. And then we can access the data attributes. But that's kind of lame, right, because there isn't anything cool we can do with it. There isn't ways to interact with this object. So let's add some methods. Remember methods are going to be procedural attributes that allow us to interact with our object. Methods are like functions except that there's a couple of differences which you'll see in a moment.

And when you're calling methods, you're using the dot operator, like L dot append, for example, for lists. So let's go back to defining our coordinate class and let's define a method for it. So so far we've defined that part there, class coordinate and an init. So we have that. So in this slide we're going to add this method here. So this method here is going to say I'm going

to define a method called distance and I'm going to pass in two parameters.

Remember self, the first parameter, is always going to be the instance of an object that you're going to perform the operation on. So pretty much by convention it's always named self. And then for this particular method, I'm going to give it another parameter, and I can name this whatever I want. I'm naming it other. And this is going to represent the other coordinate object for which I want to find the distance from my self. So here I'm going to just implement the Euclidean distance formula, which is x1 minus x2 squared, plus Y1 minus Y2 squared, and square root of all that. So that's what I'm doing inside here.

Self and other are coordinate objects. Inside this method, I have to refer to the x data attributes of each object if I want to find the difference between the 2x values from them. So that's why I'm doing self dot x here, right. If I just did x, I would be accessing just some variable named x in a program which actually isn't even defined. So you always have to refer when as we're thinking about classes, you always have to refer to whose data attribute do you want to access? In this case, I want to access the x data attribute of my self, and I want to subtract the x data attribute of this other coordinate, square that, same for y, square that, and then add those and take the square root of that.

So notice this method is pretty much like a function, right? You have DF, some name, it takes in parameters. It does some stuff and then it returns a value. The only difference is the fact that you have a self here as the first thing and the fact that you always have to be conscious about whose data attributes you're accessing. So you have to use the dot notation in order to decide whose data attributes you want access. So we've defined the method here, distance.

So this is in the class definition. Now how do we use it? So let's assume that the definition of distance is up here. I didn't include the code. But really all you need to know is what it takes. It takes a self and an other. So when you want to use this method to figure out a distance between two coordinate objects, this is how you do it.

So the first line, I create one coordinate object. Second line, I create another coordinate object. First one is named C, the second one is named 0. These are two separate objects. And I'm going to find the distance. And I want to first call it on one object, so I'm going to say C dot, so I'm using the dot notation to call the method distance on object C. So Python says this object C is of type coordinate. It's going to look up at the class coordinate that you defined. It's going to find this method called distance and then it's going to say what parameters does it

take? So it takes another parameter, right, for the other and then, in the parentheses, I just have to give it this other perimeter.

An easier way to see what happens is by looking at what this line here is equivalent to. So the third line here prints C dot distance 0 is equivalent to this one on the right. And this one on the right essentially says, what's the name of the class, dot, dot notation, what's the method you want to call, and then in parentheses you give it all of the variables including self. OK. So in this case you're explicitly telling Python that self is C and other is 0. So this is a little bit easier to understand, like that.

But it's a little cumbersome because you always have to write coordinate dot, coordinate dot, coordinate dot, for every data attribute you might want to access, for every procedural attribute you might want to access. So by convention, it's a lot easier to do the one on the left. And as I mentioned, Python implicitly says, if you're doing the one on the left, you can call this method on a particular object and it's going to look up the type of the object and it's going to essentially convert this on the left to the one on the right.

And this is what you've been using so far. So when you create a list, you say L is equal to 1, 2, and then you say L.append, you know, 3 or whatever. So we've been using this notation on the left pretty much from the beginning of class. So we have a coordinate class, we can create a coordinate object, we can get the distance between two objects.

As you're using the class, if you wanted to use this coordinate class, and you were maybe debugging at some point, a lot of you probably use print as a debug statement, right? And maybe you want to print the value of a coordinate object. So if you create a coordinate object, C is equal to coordinate 3, 4, right? That's what we've done so far. If you print C, you get this funny message.

Very uninformative, right? It basically says, well, C is an object of type coordinate at this memory location in the computer. Which is not what you wanted at all, right? Maybe you wanted to know what the values for x and y were. That would be a lot more informative. So by default, when you create your own type, when you print the object of that type, Python tells you this sort of information which is not what you want. So what you need to do is you need to define your own method that tells Python what to do when you call print on an object of this type.

So this is going to be a special method, just like init is, because it starts and ends with double

underscores. And the name of the method is underscore, underscore, str, underscore, underscore. And if you define this method in your class, that tells Python, hey, when you see a print statement that's on an object of type coordinate, call this method, look what it does, and do everything that's inside it. And you can choose to make it do whatever you want inside your definition of str.

In this case, let's say when we print a coordinate object, we're going to print its x and y values surrounded by angle brackets. That seems reasonable, right? So then from now on when you print coordinate objects, you're going to see things like this, which is a lot more informative. So how do we define this? So so far we've defined all that and the last part is going to be new. So we define the init and the distance, and let's define this str.

So underscore, underscore, str, underscore, underscore, is a method. It's only going to take self because you're just calling print on the object itself. There's no other parameters to it. Str has to return a string, and in this particular case, we're going to return the string that's the angle brackets concatenated with the x value of the object, self.x, concatenated with a comma, concatenated with the y value of this particular instance of an object, self.y, and then concatenated with the angle brackets. So now any time you have print on an object of type coordinate, you're going to call this special method str, if it's implemented in your code. Any questions? OK.

So let's try to wrap our head around types and classes because we've seen a lot today. Let's create a coordinate object, assign it 3, 4, as we have been, and assign it to variable C. We've implemented the str method, so when we print C, it's going to print out this nice three comma for our angle brackets. If we print the type of C, this is actually going to give us class main coordinate, which tells us that C is going to be an object that is of type class coordinate.

If we look at coordinate as a class, if we print what coordinate is, coordinate is a class, right? So this is what Python tells us, if we print coordinate, it's a class named coordinate. And if we print the type of a coordinate, well that's just going to be a type. So class is going to be a type. So you're defining the type of an object. If you'd like to figure out whether a particular object is an instance of a particular class, you use this special function called is instance. So if you print is instance C comma coordinate, this is going to print true because C is an object that is of type coordinate.

Couple more words on these special operators. So these special operators allow you to

customize your classes which can add some cool functionality to them. So these special operators are going to be things like addition, subtraction, using the equal equal sign, greater than, less than, length and so on and so on. So just like str, if you implement any of these in your classes, this is going to tell Python. So for example, if we've implemented this underscore, underscore, add, underscore, underscore in our class, this is going to tell Python when you use this plus operator between two objects of type coordinate to call this method.

If you have not implemented this method and you try to add two objects of type coordinate, you're going to get an error because Python doesn't actually know right off the bat how to add two coordinate objects, right? You have to tell it how to do that. And you tell it how to do that by implementing this special method. Same with subtract. Same with equals.

So if you want to figure out whether two objects are equal. And when you implement these methods in your own class, you can decide exactly what you want to do. So what happens when you add two coordinate objects? Do you just add the x values, do you just add the y values, do you get them both together, do you do whatever you'd like to do. And then you document what you've decided.

So let's create a fraction object. So we've looked at coordinate, we saw sort of a higher level car object. Let's look at a fraction object. Fraction object is going to be, is going represent a number that's going to be a numerator slash denominator. OK. So that's going to be a fraction object. So the way I've decided to internally represent a fraction object is with two numbers. And I've decided that I will not let them be floats. They have to be integers, hence the assert over here. So inside the init, I've decided I'm going to represent my fracture with two numbers, one for the numerator and one for the denominator.

So when I create a fraction object, I'm going to pass in a numerator and a denominator. And a particular instance is going to have self dot numerator and self dot denominator as its data attributes and I'm assigning those to be whatever's passed into my init. Since I plan on debugging this code maybe possibly sometime in the future, I'm also including an str method and the str method is going to print a nice looking string that's going to represent the numerator, and then a slash, and then the denominator. And then I've also implemented some other special methods.

How do I add two fractions? How do I subtract two fractions? And how do I convert a fraction to a float? The add and subtract are almost the same, so let's look at the add for the moment.

How do we add two fractions? We're going to take self, which is the instance of an object that I want to do the add operation on, and we're going to take other, which is the other instance of an object that I want to do the operation on, so the addition, and I'm going to figure out the new top. So the new top of the resulting fraction. So it's my numerator multiplied by the other denominator plus my denominator multiplied by the other numerator and then divided by the multiplication of the two denominators.

So the top is going to be that, the bottom is going to be that. Notice that we're using self dot, right? Once again, we're trying to access the data attributes of each different instance, right, of myself and the other object that I'm working with. So that's why I have to use self dot here. Once I figure out the top and the bottom of the addition, I'm going to return, and here notice I'm returning a fraction object. It's not a number, it's not a float, it's not an integer. It's a new object that is of the exact same type as the class that I'm implementing.

So as it's the same type of object, then on the return value I can do all of the exact same operations that I can do on a regular fraction object. Sub is going to be the same. I'm returning a fraction object. Float is just going to do the division for me, so it's going to take the numerator and then divide it by the denominator, just divide the numbers. And then I'm defining here my own method called inverse. And this is just going to take the inverse of the instance I'm calling this method on. And so it's going to also return a new fraction object that just has the denominator as the top part and the numerator as the bottom part. So then we have some code here. So that's how I implement my fraction object.

So now let's use it and see what it gives us. A is equal to a fraction 1, 4. This is going to be 1 over 4 for a. And b is going to be 3 over four. When I do C, notice I'm using the plus operator between two fraction objects, right? A and b are fraction objects so Python's going to say, OK, is there an underscore, underscore, add, underscore, underscore, method implemented? It is and it's just going to do whatever's inside here. So it's going to say self dot numerator plus other dot denominator. It's going to calculate the top and the bottom. It's going to turn a new fraction object.

So this is going to be 4 plus 12 divided by 16, and 16 over 16. So C as a fraction object is going to be 16 for the numerator and 16 for the denominator because it's a fraction object. If I print C, it should print 16 over 16, so we can even run it, so print 16 over 16. If I print floats C, so this special method float here is going to say, is there a method that converts a fraction to a float and there is. It's this one implemented right here. So it's just going to divide the two

numbers, top and bottom, which gives me 1. So it's this one here and here.

Notice I'm doing the exact same method call, except I'm doing it the other way where you type in the name of the class, name of the method, and then what you're calling it on, and this gives the exact same value here, 1.0. And then here I'm calling the method inverse on object B which is going to invert 3 over 4 to be 4 over 3. And then I'm converting it to a float and then I'm printing the value. So it gives me 1.33. So take a look at this code in more detail and see if you can trace through all of those different things and see if you can also write your own new fraction objects. OK.

So last slide. Power of object oriented programming is that you can bundle together objects that are of the exact same type. And all of these objects are going to have the same data representation and the same methods that you can do on them. And ultimately, you're going to be building these layers of abstraction. So you're going to be building on a basic object type in Python, you're going to have integer objects, float objects. On top of those, you can create lists, dictionaries. And on top of those, you can even create your own object types as we saw in this lecture today.