[SQUEAKING] [RUSTLING] [CLICKING]

ANA BELL: All right, let's get started. So today, we're going to be starting a completely new set of topics. And we'll be talking about these topics for the next four lectures. And it's a big topic. The big idea we're trying to accomplish in these next four lectures is for us to start defining our own object types. And we'll be defining these object types through these things called Python classes. So today's lecture, we'll just give you a really-- we'll just define a really simple object type. And then we'll build up from there on.

So let's take a step back and think about particular objects, really specific objects that we've been working with. So for example, we've been working with probably the number 1234. We've been working with the float pi, 3.14159. We've been working with sequences of characters like Hello, with lists of numbers. So here's a list with those specific elements within it. And we've been working with dictionaries. And here's a specific dictionary with these entries.

Now, every one of these things up here is an object. We have it in our program. We can manipulate it. We can add it to other things. We can index. We can do all these things. But every one of these objects basically has a certain type. We talked about this back in lecture 1 when I introduced types of objects.

So what does that mean? Well, in that lecture, I said that the type of an object basically tells us the kinds of things that we can do with it. So the things you can do with a number are going to be different than the kinds of things you can do with strings. And we've been seeing this since that lecture up until today.

Today, we're going to see how we can create our own object types. So to do that, we have to understand the following thing. And this is something I'll keep repeating today. So once you decide to create an object type-- every one of these objects, for example, has been created using some blueprint.

And when you're creating these objects, you need to think about two things. The first is what data will represent this object. And the second is what behaviors will this object have. Now, the objects up here are pretty simple, right? The kinds of data that represents this integer is-- well, there's no data, really. It's just the number itself. But it has some operations, some things that you can do with this integer.

Now, the data that represents a list is going to be different than the data that represents an integer because the list is kind of made up of a sequence of numbers or objects. And then the data that makes up a dictionary is completely different than the data that makes up the list because a dictionary has entries, where each entry has a key and a value pair. And then you have a bunch of these entries.

So the data representing each one of these objects is different. And we're going to decide what data represents the new objects that we want to create. And of course, this is something we've known from the first lecture. The kinds of ways that we can interact with these objects is also different amongst all these different object types.

So in terms of terminology, when we create an actual object that we want to manipulate, we call it an instance of a type. So this specific number, 1234, is an instance of an integer. And this specific sequence of characters-- lowercase hello, is an instance of a string.

All right, so the idea of object-oriented programming is basically that everything in Python is an object. And this we've talked about when we were introducing functions. We treated functions like objects. And what that means is that we can create new objects that have some type. So we actually create these very specific objects that we can manipulate. And we can also destroy the objects. So you can create them, manipulate them, and destroy them as you will. But each one of these objects has a specific type.

So let's talk a little bit more about the data abstraction. So once you have an object that you'd like to create-- so think of anything in the world, some something-- the two things that you need to think about are what is going to be the way that you represent the object in terms of data. And the other thing is what are the behaviors of this object. How can other programmers or other objects interact with this thing that you're creating?

So when we're creating our own object type, we have to think about these data abstractions. So let's take a more real life example. So let's say I have these two very specific cars that actually exist in the real world. So we can actually drive these cars around. We can manipulate them. They have already been created. They are actual objects.

Well, behind the scenes, these objects were created using some blueprint. This blueprint is not an actual tangible thing. It's basically some abstract notion of how to create those specific objects, those specific cars. So as we're thinking about creating our own object types, we have to think about design decisions.

If I want to create a blueprint for a car that somebody can then use to create an actual car in real life, how do I abstract the car? And as we're creating these objects ourselves, we get to make these design decisions, which is pretty cool. So if I were creating this car, the blueprint for a car, I would say, well, I'm going to use maybe the length of the car, the width of the car, and the height of the car, and the color of the car. And those four data attributes will represent a car object.

But of course, that's my design decision. If you were more familiar with cars or if you wanted to get into a more detailed description or representation, you would also have a number for how many horsepowers it has, how many doors it has, maybe how many people could fit, other things like that. But a very simple data abstraction for a car is length, width, height, and color.

So that's data abstraction, so what data represents, this object you're trying to create. Now, how about the interface? Well, in terms of the interface, we decide what are some ways that programmers can interact with the object or other objects can interact with this object.

So we could say that we could let the users change the color of the car. We could say that we can let the car make a noise. So honk the horn could be maybe one thing, one function that this car could do. And if we say honk the horn, then maybe it would print something to the screen, something like that.

And then we can have the car drive from point A to point B. Or we could have the car go in a circle. You could have the car crash another car. And all of these behaviors are part of this the interface for this particular car. But we're going to define them such that any car that we create from here on, any actual object that we create will have all of these behaviors and all of these data attributes.

So an example a little closer to home is the list. We've been working with lists so far. So behind the scenes, somebody created the data type list. So there's some code in Python that basically defines the data that makes up the list, the data attributes-- how is a list described-- and the behaviors, the procedures, the functions that a list can do.

So in terms of data attributes, well, there's many design decisions that whoever decided to create this list class could have done. How could they have represented the list? Well, they could have said, I'm going to allocate sort of a contiguous block of memory. And your elements will go in that order from the smallest memory value to the biggest memory value. That's one design decision.

Another one could be that, instead of allocating a contiguous block of memory, you could say, I can allocate memories here and there. That's OK. But then each element in my list will then be represented by two things-- the first being the value at that location. And the second could be maybe another integer or something that tells Python which memory location to go to to get the next element in the list.

So both valid design decisions-- I think Python did the second one. So that's how you represent the data that represents the list. And in terms of behaviors, well, we've already been working with lists. So we know a bunch of the behaviors that lists have. You can index into it. You can sort a list. You can append an item to the end of the list.

You can get the maximum element within the list. All of these different procedures, functions are things that you can do with lists. And we've been working with them. And we've been working with lists without actually knowing the representation-- how somebody decided to represent this class-- which is pretty cool.

So a couple more real life examples-- if we were to think about representing each of these-- so if we think about the object, an elevator, again, it's up to us to make the design decision. It's basically a box that can change floors, right? So we could represent it using the length, the width, the height, which are all floats or something like that. We can also represent it using the max capacity and the current floor it's at.

So all five of these variables together, values together, represent my elevator. And again, it's my design decision to do this. Yours might be different. And in terms of things that the elevator can do, well, we can change its current floor, which is basically saying change the value of the variable current floor to be something else. Add people to it-- maybe checking if you're at max capacity or not and maybe printing out a Warning if you're above that, removing people, things like that.

An employee is also a pretty common example of something that's typically implemented in a bunch of programming languages. So an employee, basically a person that has a salary, maybe works for Company X. So you could represent this employee using their name, maybe a string for the first name, a string for the last name, and then their birth date maybe, and then their salary, which is a float or something like that.

And in terms of behaviors, what can employees do? Well, you can change their name. You can change their salary. You can maybe activate or deactivate them as current employees, things like that. A queue at a store, also a really nice example. And it kind of goes hand in hand with a stack of pancakes.

How would you represent a queue at a store? Well, the representation isn't going to be a set of things. The representation could be something really simple, like just a list, which is fine. So maybe the list will have some strings with the names of the people who are currently in the queue at a store.

But what's going to make a queue kind of special is the way that we'll be using it. So the representation isn't super unique. It's just a list. But the way that a queue operates will be special because, if you think about the queue, the first person who comes into the queue will be the first person out of the queue-- first in, first out kind of situation.

So that means, if you make the design decision to add new people at the end of the queue-- so if I have a new person that gets added here, they're the newest person in. That means if I'm removing a person from the queue, I better remove the oldest one, which is going to be over at the first, at the beginning of my list.

So the way that you use the queue will be consistent with this idea. And then you can basically simulate the queue. And the stack of pancakes is very similar. If you think about pancakes, the first one you made is the last one you eat. So it's a first in, last out kind of situation. So that means that we can still represent a stack of pancakes using a list.

So the representation, the data representation for a stack of pancakes, will be the same as a queue, except that the behavior will be different because if I just made a new pancake and it goes at the end here, the newest one that I made is the first one that I'm going to eat. If I add a pancake to the end of my list, I'm going to remove the pancake that I want to eat from the end of my list as well.

OK. So the idea of object-oriented programming and the reason we're doing this is because now we're bundling basically data and behaviors into one thing. And so we can create all of these objects that have the same type that all are going to function in the same way. We know they're going to be consistent, right? They're going to be consistent in the data that represents them and consistent in the way that we use them right. We know for sure that the queue is going to be a first in, first out kind of situation.

And the way we're going to implement this is using these things called Python classes. And the reason we create these Python classes is to make code that's very nicely reusable. We can create really simple Python classes that we'll see today. And then we can build upon these Python classes to create more complex classes, which we'll see on Wednesday.

But the big idea here-- and this is something that I was a little bit confused about when I first started learning about object-oriented programming-- is you get to be in charge of the design decision. So you get to decide what data represents the class. And you decide what behaviors represent the class.

So if you wanted to say that-- you represent a queue using a list-- first in, first out-- if you add items to the end, you remove items from the beginning. That's one design decision. Another design decision could be, well, you still represent it as a list, but new items get added to the front. But to be consistent with the idea of a queue, that means you remove items from the back. And then the behavior is the same. We're implementing a queue no matter which one of those design decisions we've made.

OK. So as we're going through today's lecture, I want to make a note of a couple of things. So I've got these little tabs up at the top here. We're going to be basically switching our brains a little bit today. We're going to be defining a Python object.

So we're going to be writing code that tells Python, hey, I am telling you I would like to create this object type. This is the data that represents them, represents it. And these are the behaviors that represents it. So that's us implementing the class, so telling Python that we are now creating and telling you what an object of this type is and does.

And the other thing is, once we have a definition for this object type, we're going to actually use the type. We're going to create new objects of this type. So when we're creating the class, when we're telling Python that an object like this exists, we're deciding the name of our class. We're deciding what data abstracts it. We're deciding what behaviors we can do with it. So if we think about the list, we haven't actually seen the code to do this. But someone wrote code to define this list class.

Now, using the class means that we're assuming that this code already exists. And you're just creating a whole bunch of objects of this type. So we've been doing this definitely, right? If we think about the list class again, here for example, we created an actual object that we can manipulate. L is equal to 1 comma 2. We've also created L is equal to 3 comma 4, comma 5, and all these things. We're basically creating these instances that we can manipulate and use in our program to achieve something useful.

And today, we're going to see how we can do both of those things. I want to draw a little parallel with functions because it's going to feel very similar. And with functions, when we were defining a function, we were telling Python that I would like to abstract some code that does something useful using this class-- using this function definition.

So we were writing the definition for the function in this abstract way. We didn't actually run the function at that point. We just defined it. And so when we define a class, that's basically what we're doing. We're telling Python that we're creating this object that bundles data and behaviors together.

When we create an instances of this data type that we're going to define, that's kind of like we called the actual function that we defined. So when we called the function, we were now doing something useful in our program. We said, here are some actual parameters I want you to run this function with. Now, tell me what the output is. And that's exactly what we're going to do when we create instances of the data type we're defining. We're now creating actual objects that we can manipulate and use in our class.

OK. So the object we're going to create in today's lecture is a coordinate in a 2D plane-- pretty simple, pretty mathematical. So before we actually write the code, let's think about what it actually means to put a coordinate in a 2D plane. So we're going to think about if we had a bunch of instances, if we had a bunch of coordinates in a 2D plane, what do they look like? What kind of data are we interested in grabbing from these instances? What are some things we can do with it?

So here, I have a point in my 2D plane. So if we think about how we look at this coordinate, well, we know how far away the coordinate is on the x-axis. And we know how far away the coordinate is on the y-axis. So that's one instance of a coordinate object.

Now, let's say we had another one. Here's another dot in my 2D plane. Again, this dot will also know how far away it is on the x-axis and how far away it is on the y-axis. So one reasonable data abstraction for a coordinate in a 2D plane could be to say I want two numbers, one representing how far away it is on the x-axis and one for how far away it is on the y-axis.

That seems pretty reasonable. I don't care about color, even though I colored these things. But you can imagine making a cuter version of this coordinate object that also has a color associated with it. So the data that will represent my point in a coordinate plane, in a 2D coordinate plane is just two numbers-- one for the x, one for the y.

Now, what are some things that we can do with these coordinate objects? Certainly something really simple we can do is to say, well, one of these points-- the orange one, for example-- tell me how far away you are on the x-axis or tell me how far away you are on the y-axis. So those two commands could return something like 3 for that's how far away that point is on the x-axis or 4 for how far away it is on the y-axis.

Those are pretty simple things to do. One more interesting thing to do is to say, well, hey, you, orange point right over there, can you tell me how far away you are between the green point? So that would be the Euclidean distance between these two points. And we're going to write a code that figures out how far away one coordinate object is from another coordinate object.

All right, so let's start defining this class coordinate. You can see here, this is the code that implements the class. So this will tell Python that we are now creating this object type coordinate. So we're not using it yet. We're not creating any objects, any object instances. We're just telling Python that we'd like to create this object type.

So we start with the keyword class. In parallel, we started with the keyword def to define a function. Then we say the name of our object type. So this will be literally the type of the object, so coordinate, just like we had list and float, all those things. This will be of type coordinate.

And then in parentheses here, we say to the parent of this class. So usually, we say object until two lectures from now when we're going to see what happens when we put something else in there. But when we put object in the parentheses there, we're telling Python that anything a generic Python object can do our object can do as well.

So something really, really basic is saying that I'm going to create this object in memory and assign a variable to it so that I get a handle for that object using this variable-- something super basic. Any Python object has this ability. And ours will too because I've put this object in the parentheses here.

All right, so now, we've told Python we're creating a data type called coordinate. What are we going to fill in the body of this class? So the things we need to fill in are going to be our attributes. Now, again, what makes up an object? Two things-- the data that you want to represent this object with and the procedures, a.k.a. functions, a.k.a. behaviors that you'd like this object to have.

So the data will be two things. We decided that we're going to represent a coordinate using two numbers. Now, what about behaviors? Behaviors will essentially be functions that work with objects of this particular type. So we're going to define them as functions. But we're going to define them in a really special way that tells Python you can only run this function on an object of type coordinate, which makes sense.

I would not like to find the distance between two integers that's just subtraction. Or I would not like to find the distance between two dictionaries. What does that even mean? So distance method, that we mentioned is one thing we'd like to implement, will only work with objects of type coordinate.

So these special functions are actually called methods. And I'm going to use this term a little bit today. Hopefully, you get used to it. And then from next lecture on, I'll just use the word methods to refer to functions that only work with objects of this type.

So we so far, in the previous slide, had class coordinate object. Now , what is the next thing you have to do? So the next thing you always have to do, when you tell Python you're creating a new data type, is to tell Python how you want to construct this data type, kind of a constructor function.

And the way we do this is by defining-- so you can see we're defining it like a function, ef. But we're going to define a function that has a special name. And the name is __init__. So that's the name of this function. And you can see it's a function, def name, and then parentheses. And there's a bunch of stuff in the parentheses.

The first thing will be this thing called self. So already, it's going to be a little bit different than regular functions. Now, I'm going to-- this is not the only time I'll explain self. I'll explain it throughout this lecture. But the basic idea of self is that it's always going to be the first parameter of a method, a function that only works with an object of this class-- of this type.

And the reason why we have it here is because all we're doing here is telling Python that we'd like to create this object type. We don't have an actual object to manipulate. I haven't created an actual object yet. I'm just telling Python I'd like to create this object. So if I don't have an actual object created yet, I need some way to refer to an instance without actually having one yet.

And that's what the self is doing. It's basically a variable that tells Python that this is an object of-- that this is a function that only works with an object of this type. And I'm going to use this variable, self, to refer to this object, myself, my data attributes, and my methods, and things like that. So it will become clear. There will be many examples. But for now, it's basically a way for us to refer to an object of this type-- an instance of this type without actually having created one.

Anything after self is basically parameters you'd like to create this object with. So for us, it doesn't make sense to say, create this coordinate object without actually initializing its x and y values. When we put a coordinate object in a 2D plane, I would like to put it in that 2D plane. So it needs an initial x and initial y value.

So these parameters here will tell Python you need to pass in a value for x and y when you create your object. And then the body of this it will have whatever you'd like, whatever code you'd like to initialize your object. Yes, question?

STUDENT: The way you put the underscores, is that part of how you write it?

ANA BELL: The underscores is part of how you write it. So you have to have __init__. Yeah, it's a special function. We'll talk about them next lecture. It's called a Dunder function, double underscore function, Dunder.

OK. So the body of this function can contain a bunch of initialization code. So anything you'd like to initialize when you create an object of this type, that's what you stick in here. Usually, most of the time, 99% of the time, you want to initialize the data that makes up your object.

So the data we decided makes up our object is how far you are on the x-axis and how far away you are on the y-axis. So here, this data that I want every single one of my objects to have-- a value for x and a value for y-- is initialized using self.-- so self. a variable named x and self. a variable named y.

And the self. before these variables distinguishes these variables, x and y here, from regular variables. If I were to just say x equals xval and y equals yval, x and y will just be regular variables. As soon as my init function terminates, those variables are gone.

But because I've got self.x and self.y, this means that these values, x and y, will persist throughout the lifetime of my object when I create my actual object. And every single object I create will have their own x and y values. Question?

STUDENT: Does it have to be different to the xval? So kind of self.xval or--

ANA BELL: Yeah, good question. Does this self. Thing have to be different? It does not have to be. So you can have self.xval equals xval and self.yval equals yval. The reason I did it here is to showcase that they actually do not have to be the same. Yeah, they are completely different. So self.x is different than xval. We just happen to be assigning this value to be whatever is passed in.

OK. So a little bit of, again, just explaining what the self is in the context of a blueprint-- so if we think about a blueprint in real life-- so here, I have a blueprint for a room that I might want to create. I don't actually have this room created yet. It's just an idea. But what I know is that I'm going to use this blueprint to have a room that contains two chairs, a coffee table, and a sofa.

So in this blueprint, I don't have actual rooms that I've implemented this thing in. I don't have actual rooms where I've put two chairs, a coffee table, and a sofa in. It's just an idea. But self is the way that a blueprint accesses its attributes. So if I say self.coffeetable, that means if, in the future, I have an actual room, self.coffeetable means I'm referring to that room's coffee table.

So the self is a variable that we use to refer to data or to attributes for a blueprint when I don't have actual rooms created. But once I create instances of rooms-- so for example here, I have something called livingroom created. So I've taken my blueprint. And now, somebody asked me to create a room with this blueprint.

Now, I no longer use self because I have an actual room in hand. So now, I would refer to coffee table in this living room as livingroom.coffeetable or livingroom's coffeetable, no longer self's coffeetable. So self is only used in the context of my blueprint.

And to bring the last point home, the idea that with the blueprint you can create many different instances, well, here's a living room that I've applied my blueprint to. And here's another living room, completely different room that somebody asked me to use my blueprint for to create it-- different chairs, different coffee tables, different colored things. These are all different instances that I used my one template, my one blueprint for the room.

So when we're defining a class, we don't have actual objects. Again, that's just a really big idea here. We're just telling Python, I'd like to create this object and this is what it looks like. I'm bundling this data with these behaviors together. But I don't have actual objects of this type created yet.

So let's actually create some objects. The code that does this is as follows. So I've put the definition for my class, the constructor, the init method for my class up here, just to remind us what it looks like. And with that code, we can now start to create actual objects that we can manipulate. So when we created something like L is equal to square bracket lists 1 comma 2, now I'm creating these actual coordinates in my code using my blueprint.

So the way we do that is we invoke the name of our class. So you say coordinate-- that's what we named it, right? That's our data type. And here, I'm passing in every single parameter except for self. So I initialized a coordinate object using xval and yval. So I need to put in two parameters here for xval and yval.

And self actually becomes this thing that I just created, this object. So coordinate 3 comma 4 is now an object that's being referenced by a variable named C, which is why I'm not passing in self. So it's kind of weird to think about. But now, I have one object in memory. It's referenced by name C.

And on the next line, I have another object in memory. Again, I've invoked the name of my class, coordinate. This particular object, x value will be 0 and y value will be 0, so different than the one I just did on the previous line. But it'll have the same structure.

So they will both have some x and y-value. They'll just be different from each other. But they'll both have x and they'll both have y. The one I've named down here is going to be origin. So I've got two objects of type coordinate. One is referenced by C, by name C. And the other one is referenced by name origin.

So now that I have these objects in hand, I can access any of their attributes. And Python will grab for me the attribute of that particular object. So here, I've got this thing called dot notation, which we've seen before. And I'll explain it again in a couple of slides.

But the dot notation tells Python to access the x data attribute of object C. So this will grab for me the x value of C, 3. And the next line will grab for me the x value of origin, 0. And this is all made possible because x-- and we could also access y-- x and y were defined in the class definition using self. If I didn't use self., those would just be variables. And as soon as I created my object, they would have gone away because that function had terminated.

But in order to have these variables, x and y, persist throughout the lifetime of my object, I've defined them using self.x and self.y. So any object I've created that's of type coordinate will have some value for x and some value for y. So we can access that value through this notation. Does that make sense so far? Is that all right? OK.

So we're going to visualize this in a slightly different way. So the exact same code as on the previous slide, we're now going to do it in our little memory type. So here, I have C is equal to coordinate 3, 4, exactly what I had on the previous slide. So in memory, the way you think about it is as we've been thinking about other objects. It's not much different.

We have C is our name. And it's bound to an object of type coordinate. It just so happens we define this object. But it's the same idea. I've got a name bound to some object. And this object has its own x value and its own y value. So when you evaluate c.x, Python goes into memory and says, hey, what type is C? And it says, oh, it's a coordinate object. Does coordinate object have a data attribute named x? Yes, it does, because it looks at the init. And then it says, well, what's its value? It's 3. And so it just returns that.

And so the next three lines here are slightly different from two slides ago, but very similar. a is equal to 0 creates for me a variable named a bound to the value 0, just to showcase that it's exactly the same as having a variable named c bound to this object that we created. And then when I say orig equals Coordinate(a,a), Python says, all right, well, here's a name orig, for origin.

What is it bound to? Well, it's also bound to an object of type coordinate. And it's an object we defined. So we defined an object of type coordinate having an x and y value. So here they are. And they're originally 0. They're set to 0 when I created this object.

So when I say orig.x, Python will look up orig. It's going to say, hey, what type are you? Oh, you're a coordinate. Do you have an x value? You do. That's what we defined in the init. Let me grab that value from you. So we're just manipulating objects in memory. Now that we've written the code to work with objects that we created, we're just creating a whole bunch of these objects in memory, and then grabbing their x values. And then we're going to get the distance between two objects in a bit.

One more way to show you that exact same code is to visualize it. So here is the code, the entire code as you would have it in a file. So you would have all this altogether. The gray box is the definition for my object type. And the blue box is me using this object that I just created. I've just separated that out just for clarity.

So when I have my gray box, there's nothing to display. It just sits in memory. And Python knows of this type of class coordinate that has two data attributes-- the things that I've defined using self., x and y. When I create c is equal to Coordinate(3,4), visualizing what we're trying to do here, here, I've got this object whose name is c. And it's at 3 comma 4. And then I've got this object named origin. And its x and y values are 0 comma 0.

So because I've created these objects using the same blueprint, the coordinate blueprint that I've defined up in the gray, that means every object that I've created, c and origin, has a self.x and self.y value. It just so happens that the actual values for x and y are different between these two objects. So when I grab origin.x, I'm looking up origin and I'm grabbing its x value, 0, so just another way to visualize it.

OK. Is everyone OK with these data attributes? All right. So now, let's add a method. So a method, remember, is just a function that works with an object of this type. So the way that we tell Python we'd like to create a method is by passing in self as the first parameter.

So let's create this function named distance. If you look in the actual Python code for today, I've got two more functions-- one to get the x value of this current object and one to get the y value. But those are not as interesting. This distance one is interesting, though. So I would like to create this function that only works with an object of type coordinate. So what we've done so far is these lines up here.

So now, we've got def-- again, it's just the function. So we've got def-- name of it, distance, and then the parameters. So again, since this is a function that only works with an object of type coordinate, I need to put self as the first parameter. And this self will help us refer to the object when I call the method on it.

So if self is the first parameter, that means that this distance method will be called on self. So when I have an actual object in hand that I'm calling distance on, the self parameter will take on the value that is that object. We're going to see this in the next slide. So self is the thing that I'm calling this function on.

And then what other parameters do I want to give to this function? Well, I want to find the distance between my self, so this object that I'm going to call distance on, and another coordinate object. Now, other than maybe a docstring here that says, hey, warning, other should be an object of type coordinate, there isn't really anything that enforces the type of other when you make a function call-- or when you make a method call.

So you can call this distance method with other being an integer, which is not an object of type coordinate. The code will run, but will immediately crash because of what's going on inside. So the only way this code will work is if you're calling it on an object of type coordinate for the other.

So the reason for that is because, well, when we think about grabbing the distance between two objects that are coordinates in a 2D plane, we take the difference between the x values, square that, take the difference between the y values, square that-- Pythagoras-- add those two together, take the square root.

So if I'm calling this distance method on an object of type coordinate, i.e. self, how do I grab my self's x value? Well, I just say self.x. My x value, what is it? And then I would like to subtract that from the other coordinate object's x value. What's my other coordinate object? It's the thing that I'm passing in as a parameter. So grab their x value.

So if I take self.x minus other.x, Python will grab my x value, subtract it from other's x value, square that. We do the exact same thing with y. We grab my y value, subtract it from other's y value, square that. And then the rest is just Pythagoras. Add those two and take the square root. And you take it to the power of 1/2.

And this function is just a regular function other than the self being the first parameter and us working with data attributes of my self and potentially other parameters. But you can see it returns a value. It has the def, the name, and things like that.

So the way we're going to use this method that we just wrote is using the dot operator. Just like we accessed a data attribute of an object that I created, I can access a procedural attribute, i.e. a method of an object I just created. So we use the dot operator for this. The thing before the dot is the object I would like to call the method on dot the name of the method I'd like to call. And in parentheses, it's just a function. So I need to give it any parameters this method expects.

Now this, should look very familiar. We introduced dot notation when we worked with lists. Remember that? And I said, when we work with a list, you, for now, have to remember why we use this special way of writing this function. But it was the same idea. The thing before the dot was the list I wanted to apply the function to.

So my_list is the name of a list variable. I wanted to apply the function append. And it happened to take an integer as a parameter. And same with sort here is also another one. But this one didn't take any parameters. But it's the same idea, the dot notation.

So in terms of our class, here, I've got two corded objects. And I've got a dot notation being used here to find the distance between one object and another one. So the thing before the dot is an object I would like to use the distance method on. Pick one of them, c.distance, the name of the method I would like to call. And in parentheses, I've got another coordinate object, orig. So here, I am using the class. And I've got actual values, actual objects that I'm manipulating-- c and orig.

So this might look a little bit weird. But when we actually call the function, remember, we omitted self-- when we omitted-- sorry, we omitted self when we made this function call. But that's because self implicitly becomes the thing before the dot, the thing you're calling this method on.

So let's visualize that in our memory. So here, I've got my class definition for a coordinate. It has some data attributes and some procedural attributes. I've got these two objects being created. c is this object of type coordinate. orig is this object of type coordinate. They've got different x and y values. But they both have some x and y values.

When I make a function call to c-- sorry, a method call on c, Python says, all right, let me look at this thing before the dot. What is it? It's an object of type coordinate. Then it looks at the method you're trying to call, distance. It says, hey, does coordinate have a distance method defined? Why, yes, it does. We just wrote it.

And then it says, all right, well, let me call this distance method. It's going to set self as c, the thing before the dot. And any other parameters will be set in order to whatever is being passed in here. So orig will become the other parameter from my definition for that function.

So this is just the conventional way of calling methods. And it's the way we've been working with lists, and dictionaries, and things like that. So again, we've got some object, the thing before the dot, some method to run. And when we call it this way, the thing before the dot becomes self in our class definition, in our method definition. And then all the other parameters become assigned one by one, except for self.

Now, to demystify this, I would like to show you what this is actually equivalent to. So we can run the function, the method that we defined by actually passing in a value for self, if this is clearer to you. So in that case, the thing before the dot cannot be an object because if it is an object of the type coordinate, then Python will say, well, this is the object I'm running the distance method on.

So to demystify this, you can actually invoke the name of the class, the object that you're trying to create, the name, the data type, coordinate. And then Python says, oh, I see, you're calling the name of the class. It's not an object. So then what do you want from me? The thing after the dot says, I would like to run this method on you.

But now, it needs all the parameters in the parameter list, including self. So here, I would have to give it explicitly c comma 0 instead of just 0 because the thing before the dot is the name of my class, not an actual object, like it is on this side. So this is actually the conventional way to do this. This is the shorthand, the Pythonic way to do this. But this hopefully demystifies the self deal and the way we actually set that first parameter to the thing before the dot. All right. Yes, question?

STUDENT: When you were going to do the first one, you had more than one parameter [INAUDIBLE] 0 comma--

ANA BELL: Yeah, exactly. If there's more parameter, just pop in those extra ones with commas, just like a regular function. So this dot operator basically accesses either our data, c.x, or our methods, c.distance, or whatever, or whatever method name we have.

So that's it for today's lecture. Next lecture, we're going to build on this coordinate object by creating circles. And then we'll create some fraction objects. And we'll look at some other objects that we can bundle together. OK.