[SQUEAKING] [RUSTLING] [CLICKING]

ANA BELL: All right, let's begin today's lecture. So last class, we began our adventure with creating our own data types. Today, we're going to start off with a little bit of a recap, just to remind you of some of the details about creating our own data types. And then we're going to build upon that coordinate class we started on last class. We'll build a circle class. And then we'll build some fraction data types.

All right, so the first thing I'd like to mention is to remind you guys about writing code from these two different perspectives. So just like when we wrote functions, we were writing the definition of the function, telling Python we have this function that we're defining abstractly and this is what it does. And then we were calling the function later on in the program many, many times.

Well, the same thing exists-- the same idea exists now that we're creating our own data types. We have to write code from the point of view of somebody who's implementing the class, so deciding all of these details that goes into creating the class itself, versus somebody who's just using a class that's already been written, where we create instances, a bunch of different objects that just happen to be this data type.

So when we implement the class, what were some of the things we did? Well, we're telling Python that this object now exists. We're telling Python the name of the data type that we're creating, something we choose. We're making these design decisions where we decide what attributes make up our class.

So the attributes are either data, like the properties-- what are the variables that make up your object-- and the behaviors through methods. So that's implementing the class. And then when we're using the class, we're now saying, all right, let's assume that this class definition exists.

There's this object that has these behaviors and these data attributes. Let's now create a whole bunch of objects that are of this type. And this is when we're creating these instances, and then manipulating all of these instances by running methods on them, things like that.

So when we're implementing the class, this thing on the left-hand side, we're basically telling Python, in abstract terms, what are the common property and behaviors of our data type. And then when we're using the class-- the thing on the right-hand side here-- we're creating actual objects with very specific values for their data attributes that we can manipulate in different ways.

So let's remember this coordinate class that we wrote last lecture. This is not new, but I will just go over it real quick. So first line here tells Python we're creating a new data type. Its name is coordinate. And this keyword class tells Python we're creating the data type. The parentheses here is object, which stands for the Python object data type. So it's something really generic.

And this, in the parentheses here, is the parent of our class. So anything that a regular Python object can do, the very basic things, our class can do as well. Last lecture, I mentioned an example of such a basic thing is to take a variable name and assign it to an object type that we create.

The very first method that we should write for a new data type that we create is the init method. And this I called a dunder method because it starts with double underscores before the init and ends with double underscores after the init. And that's the actual name of this method, __init__.

So this method is like a constructor for the class. It tells Python how do you create an actual object of this type. So it's a function. It's just a function that works only with objects of type coordinate. So as a function, it takes parameters. You can see it takes three parameters here-- the self, the x, and the y.

Now, when we're actually creating objects of type coordinate, we only pass in parameters for everything other than self because self is a variable name that we use to describe having an instance of the class without actually creating one yet because remember what we're doing here in this definition. We're telling Python that this object type now exists. We're writing it as we speak.

But we don't have an actual instance to manipulate yet. This is just the definition. And so the self tells Python that, when we're writing this code, we're going to use the self variable name as a formal name to be able to run this method on. So we're going to see in the next slide exactly what maps to self when we run it. But that's what the self means inside the parameter list here and here.

And then beyond that, we use self within the init method to tell Python which one of these variables are actually data attributes versus which of these variables are just plain old variables as we've been working with. So any variable that's defined using self.-- so here, I've got and self.x and self.y-- are data attributes. So that means any object I create that's of type coordinate I know will have a variable x and y associated with it because I've defined these x's and y's using and self.x and self.y.

Now, in the last lecture, I actually had these parameter lists-- the parameters in this list here be different than x and y. I think I had xval, yval. And then I had self.x equals xval, where this x here to the right of the equal sign is the x from the parameter list. So in that sense, it doesn't matter what these variables are in the parameter list. They're just going to be the same over here on the right-hand side of the equal sign. But the actual parameters-- sorry, the actual data attributes are self.x and self.y of my object.

OK. So then we had one method that we wrote last lecture. It was called distance. And it took two parameters-- so the first one, of course, is self. And this self represents the thing, this object that you're going to call the method on. I don't have that object yet, so I'm just calling it self for now because this is the class definition. And then this other parameter is some other coordinate object that I'm going to run this method on.

So the body of distance says, all right, well, how do I find the distance between two points in the 2D plane? It's just Pythagoras, right? So that means grab the x value of 1 of my points, subtract the x value of the other point, square them. Same with the y's, square them, add them, take the square root.

So what's the x value of one point? Well, one of the points is going to be the thing that I'm calling the distance method on, self. So I grab the x value of self using this dot notation, self.x. And then what's the x value of this other coordinate point? Well, it's called other in my parameter list. So I'm going to grab the x value of other, again using dot notation. And then we just do the math. Yes?

STUDENT: Can you only call functions on a class that were defined like this?

ANA BELL: Yeah, so you can make methods. You can make methods for a particular class.

STUDENT: Right, but you can only call those functions that you define? You can't call another functions?

ANA BELL: Exactly, yeah. Yeah. Yeah?

STUDENT: Is there a way we could define a class as something other than an object?

ANA BELL: A class as something other than an object?

STUDENT: Right, like say we put coordinate and then object, then you put something else.

ANA BELL: In the parentheses? Yes, we can put other things in the parentheses. So that's actually what Monday's lecture will be about. In that case, the thing in the parentheses becomes the parent of the class that you're currently writing. So I won't go into too much detail. But to have this other object as a parent means that everything that that object can do automatically your object can do as well.

And then on top of that, you can decide a bunch of additional stuff your new object does. But in a sense, your coordinate object is a whatever this thing in the parentheses is. And then it can do a bunch of other stuff as well. Yes?

STUDENT: Similarly, how you can copy and paste the square brackets?

ANA BELL: Yeah?

STUDENT: How can you do that?

ANA BELL: Oh, to make a copy of the object? Or--

STUDENT: To make a copy of the class.

ANA BELL: Oh, so here, you can't make a copy of the class here specifically because we're just defining the class. But when you're creating coordinate objects, then you could define a method that copies your object into another object. So in essence, it would return a new object of type coordinate with whatever parameters you'd want it to do. So yeah, all of these things are possible.

So let's add one more method to this class. Let's call it to_origin. So this distance method, just to remind ourselves, returned a number. So it just took the difference between these two points and it returned a number, just how far away they are. But this to_origin method is going to do something slightly different.

Essentially, what I'm going to have this method do is to take my point from wherever it is in my 2D plane that it has been initialized to and say I'm going to reset it back to the origin. So to do that, all that means is I'm going to make its x value and its y value be 0. So I can manipulate the x and y data attributes of this particular object to be whatever I want them to be. So I can reset them both to be 0. So if I ever call this method on an object whose x and y values are something other than 0, they'll be reset to 0.

So let's actually run the code that we just wrote. So here, I've got two coordinate objects being created. So the beauty of writing this class for us is that now we can create as many coordinate objects as we'd like. They all will have an x value and a y value associated with them. It's just that the specific values for x and y will be different. So here, I've got a coordinate object with x 3 and y 4 being created. And it's going to be bound by the variable name c and here a coordinate object with x and y values both 0 bound to a variable named origin.

So then I can use this dot notation that we talked about last lecture to access either data or to run methods on the object. So in this printf statement here, I'm using dot notation on c and origin to grab the x values of c over here and origin over here.

And then I'm running the distance method on c. So remember, dot notation says the thing before the dot is going to be an object. The thing after the dot is going to be the method name that can run on this object of type whatever this is, coordinate. And then in the parentheses, it's just a function. We just pass in all the variables that that method expects. So c.distance will print however far away it is, 5, because those are nice numbers.

So then if we run this function that we just wrote, .to_origin, this function, just to remind you, doesn't actually return anything. It just resets the variables x and y for that particular object back to 0. So in here, when I call this method here-- again, dot notation, the thing before the dot is an object. It's c. Its x and y values are currently 3 and 4.

But after I run this function-- it returns none, by the way-- its x and y values will be changed to 0 and 0. So if I look at my code here-- so here's this printf statement. So c's x value is 3 and c's-- and origin's x is 0, fine. And then I've got these two calls here-- so c.to_origin. I'm making this function call. Before the function call, c's x is 3 and c's y is 4. And after the function call, you can see c's x is 0 and c's y is 0. So I'm literally changing the x and y values of this object, c.

All right, so questions so far? So far so good? Hopefully recap? OK. So again, sort similar to the first slide we started with, so we've got this class coordinate object. The class name is our type. So this object I'm creating is of type coordinate. We're defining the class in a generic way, in an abstract way.

So we have to use the self variable either in the parameter list to tell Python what's the thing before the dot going to map to-- well, it's going to map to self in my parameter list. Or we use the self to tell Python what the data attributes of this object are. So anything defined with self. some variable name will be a data attribute. That's common across all the instances I create of this type.

When we create actual instances, that's when our blueprint, our abstract definition now gets put into use. And now, I'm creating actual objects that I can grab x values from, change x values from, get distances between other objects, and so on, and so on.

OK. So what I'd like to do next is to take this coordinate class and build a circle class with it. So this comes hand in hand with the idea of when you're deciding how to create a class, you get to make the design decision. So when the Finger Exercise for Monday's lecture-- today's Wednesday, yes. Monday's lecture-- you guys had to create a center-- sorry, a circle class.

But the way we defined the circle class in that Finger Exercise was basically by that circle's radius. That's the only way we abstracted that circle. But now, in this lecture, we're going to make a different design decision and say that a circle will now be defined using two things.

The first is the radius. So I'm going to say that that's an integer. And the second is going to be the center of the circle. So as in the picture there, I'm going to say that a circle is based on the center and this radius. And the center is not going to be a float. It's not going to be a tuple. It's not going to be an int. It's going to be a coordinate object, the data type that we were just writing.

All right, good. It's not a secret. I didn't lower my voice on purpose. I just wanted to let that sink in. So one of our data attributes for the circle class is a coordinate object. So we're using an object that we just wrote to create a more complex object, a circle.

So here's my class definition. The data type is called circle. Again, it inherits all-- the parent of circle is just a generic Python object. First method we have to write is the init method. First parameter is self, so this thing that I'm creating right now. And I say that to create a circle, I have to give it a center. And I have to give it a radius.

The data attributes of the circle class-- so the two attributes that make up my circle are this self.center-- so the center variable here-- and self.radius. So these two things together make up our circle object. And initially, in the init method and when we construct our object, we're just going to set these two things to be whatever is passed in as parameters in the constructor.

All right, so what I'd like to say is that this center parameter will be a coordinate object and radius will be an int. Now, notice in this code, I'm not actually enforcing this. I could create a circle object by just passing in two strings. At this point, this code doesn't care. Nowhere am I enforcing the fact that center is a coordinate and radius will be an int. But that's just something that we know.

So then when we create the actual object down here, my_circle is going to be a variable that's bound to my circle object. So here, I'm invoking the name of my class, circle. And then what are the two parameters I'm passing in? Well, the first one, I said right up here that it should be a coordinate object.

So center is a variable name. And what is it? Well, I had to create this coordinate object. So I'm just invoking the name of coordinate, this class that creates for me a coordinate object. And I happen to put the center of the circle at 2, 2. Yeah, OK.

So this center thing is a coordinate object. It's not a tuple, or a float, or whatever. It's a coordinate object. And then the radius of this circle is 2, an int. Everyone OK with that? OK.

So what I'd like you to do next is to modify this init method just slightly, just to show you that the init method doesn't just always set the data attributes and it's done. It can do a lot of initialization code. One of the more important things it can do is to try to enforce the types on the parameters here.

So what I'd like you to do is add to this code to check that the type of center is a coordinate and the type of radius is an integer. And only if those two things are true, then do you set the two data attributes. And otherwise, raise for me a value error. So that should be around line 48.

OK. Does anyone have some code for me? Yeah?

STUDENT: In type of center does not equal coordinate [INAUDIBLE]

ANA BELL: Not equal coordinate-- Yep, so that's raise value error.

STUDENT: Value error.

ANA BELL: Yep. Cool so that takes care of one. Yep?

STUDENT: And they're the same per radius, but with the--

ANA BELL: Yep. So if the type of radius, because that's the parameter passed in, not equal to int, raise value error. So if we drop into any of these if's, then we immediately raise the value error. The code doesn't complete. And then only if we didn't drop into this one or this one do we then go on to create my object.

So then here we are. These two lines here will succeed. So there is no error raised or anything like that. But then this line here raises our value error because we tried to create a circle where the center is an integer, obviously not a coordinate object. And then here, again, we raised a value error because we tried to pass in a string as the radius. Any questions about this code? Oh yeah, sorry.

STUDENT: So [INAUDIBLE] it's very important for these two statements [INAUDIBLE]?

ANA BELL: Yeah, so it's important to place them before you actually create the object because you don't want to create it unless-- yeah, unless everything's appropriate.

OK. So now, let's add one more useful method to our class, circle. Now that we've defined a circle using a center point and a radius, we can add this little function that checks if another coordinate object is inside our circle. So again, I'm not going to be able to enforce that this point is a coordinate object. But you could do it in the docstring, or you could do a check, or something like that. But again, we're just going to assume the user using this method is going to follow the rules.

So how is this method going to work? The idea here is that we're going to use the center, which is a coordinate object, and some other point, p, wherever it may be. What we're going to do is we're going to say, what's the distance between this point and the center of the circle? If it's greater than the radius, we know the point is outside the circle. If it's smaller than the radius, we know the point is in the circle.

So this code is just enforcing that. So we have just a simple return statement that's going to run the distance method. This is a method that we wrote back in the coordinate class. That's fine because you know what? Point is an object of type coordinate. And self.center-- so the center of this circle object I'm trying to manipulate to tell if another point is inside me or not-- is also a coordinate object.

So why not? We already wrote the code that calculates the distance between these two points. So let's call it. So here, I've got the thing before the dot, a coordinate object, dot notation, the method I want to run on this coordinate object. And then in parentheses, this is another coordinate object. So this will just tell me some number for how far away these two points are. And all we do is return whether that number is less than the radius. Does that make sense?

And again, this only works if point, the thing that's passed in here, is a coordinate object. Otherwise, this code will fail because it's going to try to pass in-- it's going to try to run the distance method on a string, for example. And of course, the string doesn't have a distance method.

So down here, these two lines are exactly as we had before. We create a circle object whose center is at 2, 2 and radius is 2. And then I've got another coordinate object down here. It's at 1 comma 1, so clearly within the circle. So that print statement will then print true. So that's just basically what I wrote. This is a coordinate object. This is the method. This is another coordinate object.

All right, so let's run it. So this is exactly the code from the slide. So if I run this method on a coordinate object that's 1, 1, somewhere in here, so true. And otherwise, if I run it on coordinate object here, 10, 10, clearly outside the circle, it prints false. Questions? OK. Yes, that's exactly what I said already. OK, good.

So now, I want you to answer this question. Nothing to code here, but I've got these two is_inside methods. So the first one here, is_inside1, is exactly the one that we just saw. It runs this distance method with point and self.center. is_inside2 looks slightly different, the differences I've highlighted in this box. What I'd like you to tell me-- we can do a show of hands-- is are these two methods functionally equivalent? That is, will they return the same thing given the same input? So think about it. And then I will do a show of hands.

So who thinks yes they are functionally equivalent? Given the same input, they will both return true or false? Who thinks no they are not functionally equivalent? Some? Half and half. OK. Well, let's think about what the distance method is doing. It's being run on an object of type what?

STUDENT: Coordinate?

ANA BELL: Coordinate, exactly. So in here, is point an object of type coordinate? Yes, OK. And then here, what is the parameter to the distance method? Is it an object of type coordinate? self.center is an object of type coordinate, yes. So now, let's look at is_inside2. What is the type of self.center?

STUDENT: Coordinate.

ANA BELL: Coordinate. And we're running the distance method on this object of type coordinate. And what is the object in the parameter list here? What's its type?

STUDENT: Coordinate.

ANA BELL: Coordinate. So when we wrote the distance method, does it matter which object we call the method on to get the distance between these two points? No, right? Because the distance between-- saying I want the distance between this point and this point is the same as saying I want the distance between this point and this point. It's just the order is different.

So just the way that this distance method works, it doesn't actually matter which object I call the method on, as long as they're both corded objects, which they are. Does that make sense? Is that all right? Any questions about this for those who were in the no pot? Yeah.

STUDENT: So there's no [INAUDIBLE]

ANA BELL: self.center? self.center is an object of type coordinate, not circle. So self is a circle because self is talking about me, the class that I'm currently defining. And the class I'm currently defining is a circle. But self.center-- we even wrote code-- we would like to enforce that it is a coordinate object. So we could have put parentheses around this self.center, if we wanted to, and then call the distance on that. Does that make sense? OK.

All right, so that's all I had regarding the circle class. Now, we're going to switch gears and we're going to look at fractions-- so numerator/denominator situation here. So we're going to create a new data type to represent a number as a fraction. So first thing we need to do is make the design decision-- what data will represent our fraction? So think about it. You guys tell me. What do you think? What's a reasonable set of data that could represent our number as a fraction? When you think of a-- yeah?

STUDENT: [INAUDIBLE].

ANA BELL: Yeah, a set of two things, maybe integers, one representing the numerator, the thing above the line, and one the denominator, the thing below the line. Good, that's exactly what I had in mind. What are some behaviors of fractions? You guys tell me. What things should fractions do? Yeah?

STUDENT: [INAUDIBLE].

ANA BELL: Yeah. Yes, adding them, multiplying fractions together, dividing them, inverting a fraction also something we could do, so 1 over what it currently is, things like that. All right, so we're going to write it together. The full code is actually in the Python file. So mostly, I'm going to go through the slides, just because it's incremental, so it's easier for me to talk about it. But the exact full code is already in the Python file, if you're running it later.

So the first thing we're going to do is create this fraction class. And I'm actually going to name it SimpleFraction instead of fraction because we're going to improve upon this SimpleFraction object in a little bit. So this one I'm just going to call SimpleFraction.

Like before, its parent is the generic Python object. So again, very, very simple. It doesn't do anything special yet. The first method we need to write is the init method. So how do we initialize a fraction object? Obviously, we don't want the numerator or the denominator to be empty.

So when we create a SimpleFraction object, we want the user to tell us the values for the numerator and the denominator. So those are the two parameters that I would love the user to initiate this fraction with. And then what will be the two data attributes that we had decided on? Well, numerator-- so self.num and self.denom will be the two data attributes. So self.num and self.denom are data attributes. And they're going to be set initially to whatever is passed in that constructor call.

OK. So far so good? Let's write a method that helps us multiply two fraction objects together. So we'll call it times. So this times method will be called on an object, the thing before the dot. That object, the thing before the dot, will get mapped to self. And then the thing in the parentheses, the one other parameter, will be mapped to oth.

So how do we multiply two fraction objects together? Take the numerators, multiply them, take the denominators, multiply them, you've got your new numerator and you've got your new denominator. So how do we grab the numerators of both of these objects?

So the numerator of the thing before the dot that maps to self is self.num. And the numerator of the other object that's going to be in the parameter list is the name of my parameter, oth dot their numerator, num. Everyone OK so far? Yes, OK. I saw some head nods. That's good.

The denominator will be the same. So my new denominator is just multiplying my denominator, the thing before the dot, with the thing in the parameter's denominator. So I've got my new numerator, my new denominator. And all I'm going to do is do the division and return this value. What's the type of the return here? What's this method going to return? What type?

STUDENT: A float?

ANA BELL: A float, exactly. Yeah, good. Yep, because all I'm doing is dividing one number by another number. OK, perfect. So that's what I've already said.

Now, we can define another method, plus, to add two fraction objects together. Very similar thing, except the top is going to be slightly different, right? You take the numerator of one times the denominator of the other plus the numerator of the other times the denominator of the first one, the criss-cross thing.

The denominator is the same, just multiply the denom's together. And again, we return the division-- top divided by bottom-- again, the return of this method will be a float. So even though I'm multiplying or adding these two fraction objects together, my return will be a float. Fine.

So let's run the code. I'm creating two simple fraction objects. First one is going to be accessed using variable named f1. So this one represents the number 3 over 4. Second one is accessed by variable named f2. And this one represents the fraction 1 over 4.

So now, if I access the numerator of f1, Python says, well, what's the object before the dot? It's f1. So what is your numerator? Well, I set it to 3. So this one tells me it's 3. Pretty simple. Same thing with the denominator of f1 again, it looks at the thing before the dot. It's a fraction object. It says, do you have a denom data attribute? You do. And its value is 4. So that's 4.

Now, what's the result of f1.plus(f2)? Super weird way to write it. But it's what we've got so far. So the thing before the dot is an object. It's a simple fraction object. And the thing before the dot, remember, maps to self in my parameter list because it's just a function. So like usual functions a bunch of lectures ago, we basically map the actual parameters when we run the function to the formal parameters, the things from my function definition.

So the actual parameter here for self is the thing before the dot, f1. And the parameter f2 gets mapped to oth. That's how we read that. So this is just doing the addition. So this will give me 1.0 because it's a float. Same with the the, thing before the dot maps to self. And every other parameter in the parameter list maps to everything else except for self. So this one will do 3 over 16 to give me 0.1875. OK. Everyone OK so far?

The trick here is to remember that the thing before the dot maps to self in the method definition. And then everything else maps to everything other than self. OK. I'm glad everything's OK so far because I'm going to get you to write this code here. It looks like a lot, but the first half of it is just redefining the init method for SimpleFraction.

I want you to write these two methods. And they're going to be one liners basically. So get_inverse will return something. And it returns a float representing 1 over myself. So if the input, as in this example here, if I have a SimpleFraction object representing 3 over 4, if I call get_inverse on it right here, f1.get_inverse, self becomes f1. And I would like it to return, and therefore print, 4 over 3, so 1.3333333.

That's get_inverse. And then invert is a method that doesn't return anything. So it returns none. And instead, it just internally switches the numerator and the denominator of self. So self's numerator becomes whatever its denominator is and the other way around. So when you call it, this one doesn't print anything. But instead, if we access f1's numerator and denominator, they will have been switched.

So this is down on line 133. I'll Give you a couple of moments. And then we can write it together. It should not be a lot of code.

OK. How do we write the get_inverse? So remember, you have to return something. How do we return-- yea? 1 over--

STUDENT: Does it just return over self?

ANA BELL: 1 over self? So remember, self is an object of type SimpleFraction. So we need to manipulate its numerator and the denominator, if we want to do the return, because if we just do this-- is it this one? Yep, this one here, then Python says-- oh sorry, it's trying to divide 1, an integer, by an object of type SimpleFraction. And that's the error that we get here. And it doesn't know how to do a division between an integer and a SimpleFraction. So how can we do that by working with actual numbers that are part of my simple fraction? Anyone? Yeah?

STUDENT: [INAUDIBLE].

ANA BELL: self.denom divided by self.num. Yeah, we can do that, yep, or 1 over self.num divided by self.denom. That's also fine. Yeah, but this is a little bit cleaner. So now, denom is an integer, right? Because when we create it, we pass in int. So denom and num are integers, which means that Python knows how to do a division between a number and another number, yeah. So if we run that, now it prints 1.33333 exactly. OK. How do we do the invert function-- method, sorry. Yeah? Sorry, go ahead.

STUDENT: I first assigned [INAUDIBLE] equals self.numerator.

ANA BELL: Yep.

STUDENT: [INAUDIBLE].

ANA BELL: Yep.

STUDENT: [INAUDIBLE] is equal to new_num [INAUDIBLE].

ANA BELL: Like that, yeah. Yep, that's one way to do it. Yep, so you can see now accessing the new numerator and denominator gives me 4 over 3. Any other ways that you've done it? Yeah?

STUDENT: I used the [INAUDIBLE]

ANA BELL: Yep, the tuple trick. I like it. self.num, yep. Perfect. Yes?

STUDENT: [INAUDIBLE] I just said self.num equals self.denom. But I didn't say both of them.

ANA BELL: Oh, OK, yeah. That also works, yeah. Perfect. All very valid ways of doing it, nice. Yep, so notice there's no return for this one. I didn't want to return anything. Python will automatically return none. And these internal numerator and denominators will have been flipped, perfect. Questions about this code? OK.

So let's try it out a little bit more. So here, I've got these two additions. So this is exactly what we had previously, the exact same code. What's weird though-- and you might have been weirded out by this too when we first ran it-- is I am doing operations with two fraction objects. And yet, the plus and the times methods give me floats, which is a little weird, right?

Ideally, if we're working with fraction objects, I would like the return to also be a fraction object, so I can then work with mole fraction objects later on. That's one weird thing. Another weird thing is if we then print one of our objects that we've created, f1 in this case-- we use print statements often, to debug and things like that.

If I use the print statement on an object of a type that I've created-- in this case, a SimpleFraction-- Python spits this out. It says, hey, your object is an object of type SimpleFraction at this memory location. No thank you. That's not very useful to me, right? What I'd like to know is maybe a nice representation of my fraction object, like 3/4. I don't care about what memory location it's at.

And one more thing we'd like to try to do-- this is a class that represents something numerical. So something that people might instinctually want to do is to use operators like the star, or the plus, or the slash to divide, add, multiply these fraction things. But if we run the star operator between object of type SimpleFraction times another object of type SimpleFraction, Python gives us an error. And I'll even show you the error, so here.

So here, I am printing my object. So it spits this out, which is fine, but not what I want. This one, obviously, we've seen this already prints this out. And then if I try to multiply my two SimpleFraction objects together, it says, I don't know how to do that. So it's unsupported operand types. So the operand SimpleFraction and SimpleFraction are not supported with the star operator.

Well, no surprise there. How is Python supposed to know how to multiply two SimpleFraction objects together? Right before I even ran this program, it didn't even know what a simple fraction object was. So we need to tell it all of these details. And we will do just that.

So all of these operators-- print, len that we've been using, star, add, less than, greater than, even the square brackets to index into something, these are actually shorthand notations. They're really common operations that you want to do. And Python lets you use these common operations instead of writing these really verbose function names.

But really, behind the scenes, all of these shorthand operations actually run a method. Again, not a secret-- I'm not lowering my voice because it's a secret, but it's just really cool. So all of these operations, like the multiplication or the print statement, just gets replaced with a method.

And the method names look like this. They are dunder methods, just like the init method was a special method that Python ran when something special happened, like you're creating an object. Well, when something special happens, like you're using the plus operator between an object of your type and something else, Python will also run this special dunder method behind the scenes.

And same here, if I want to multiply my object with something else, Python will run this special dunder method behind the scenes. If I want to print an object of my type, Python will run this special dunder method behind the scenes. Even something like casting, if I want to cast my object to a float, or to a string, or something, Python will run this special dunder method behind the scenes.

And then there's a whole bunch of other ones, even indexing into a list-- or sorry, not a list, indexing into an object of your type. So if you make an object like a queue or a stack, where you know you have a bunch of sequences of objects, you can tell Python how to index into an object of your type, into an object that's a queue.

So all of these things-- all of these methods need to be implemented somewhere. Now, most of them are not implemented in the basic Python object, except for the str. So the str method actually just prints the memory location of this object. That's exactly what we had seen by the default behavior. But none of these other ones are really implemented.

And so if you want the object that you're currently writing to work with the star, or the plus, or the double equal sign to test for equality between this object and something else, you have to write the method in your class definition. So you have to implement it to tell Python that this is what you want to do when somebody uses this special shorthand notation.

So let's start with the print because it's the most basic thing you probably want to implement when you create a new data type. For debugging purposes, you'll find yourself instinctually saying, hey, print f1 to print this fraction object to see what it looks like. And so the str method is one of the really basic things you should implement right after the init method.

So let's look at it in the context of the coordinate object. So here, I've got my coordinate object, 3, 4. And even when I print this coordinate object, Python tells me this still uninformative message that this object is of type coordinate at this memory location. OK, I don't care.

Instead, what I would really like to do is say, hey, I want to represent a coordinate object by something like this-- angle bracket the value of the x-coordinate comma the value of the y-coordinate close angle bracket. So that would be a far more informative print statement than what memory location this thing is at. So let's do that.

Well, here we are-- our coordinate object, the distance the init like we had before. And here, I'm defining my str method, so __str__, no other parameters except for self, so me calling this method on an object. And what is this going to do? It will return not print, return a string.

And the string is going to represent the thing you want to be printed out. So it returns a string, doesn't print it. However you want to make up the string is up to you. So here, I've just used concatenation of a bunch of stuff. So I'm concatenating the angle bracket with the x value of my current object cast to a string, concatenated with the comma, concatenated with the y value of my current object cast to a string, concatenated with a closed angle bracket. That's the design decision we made for how a coordinate object should be printed. Everyone OK so far?

OK. So if you want to use an f-string to make up this thing to return, totally fine. If you want to make a variable in between the return and the definition here that you just keep concatenating with, so you can concatenate it with new lines and things like that, also totally fine. At the end, you just have to return that string that represents the thing you want to be printed out.

So now, let's see what happens when we actually run the code. So here, I'm creating a coordinate object. And then I'm printing that coordinate object. Well, Python says, hey, you just called a special shorthand notation on an object of type coordinate. Let me see if you implemented the str method. It looks in the class definition.

It sees the str method implemented here. And then it runs the code inside and says, oh, you want to grab c's x value and c's y value and concatenate it with these things here. Sure, I can do that for you. And then it goes and prints this out to the screen. OK. Very cool, right? Now, we can decide how to print objects that we create.

All right, so let's try to wrap our minds around types here. So if we print this c, c is an instance of a coordinate object. It's an actual object that we're manipulating. It's not the class definition. It's not anything abstract like that. It's an actual object, like the integer 3 is an actual object. So if we print that c, Python uses the str method. Well, what if we print the type of c? Somebody tell me, what's the type of c?

STUDENT: Coordinate.

ANA BELL: Coordinate. Yeah, it's the class name that we defined. So when we print the type of c, Python says, this is a class coordinate, which makes sense because if we just replace type of c here with what it is, coordinate, we'll get the same print statement. If we just print coordinate, Python says this is a class of type coordinate. All right, so those two lines are equivalent.

And then let me blow your minds a little bit more. What if we print the type of coordinate? Well, what is coordinate? It's a type. We're defining a new type in Python called a coordinate. So coordinate is a new data type in Python. So its type is type. So everything in Python is an object, even types.

OK. One more thing-- so we've used the type of something equals something else when we checked that the type of the circle's center was a coordinate. That's one way to check for types. Another way is to use this isinstance function, just as an aside. So you can check that c is an instance of type coordinate by using this isinstance method. And this will tell you true. And just to draw a parallel, you can say isinstance 3, int. That would also say true because 3 is an object of type integer. So it's just another way to check for types.

OK. So the remainder of this class, I would like to go back to our fraction class and make it better. Now that we know dunder methods, let's implement a whole bunch of dunder methods to help us and people who want to use our class use it in a more efficient way. So we're going to implement the star operator, the plus operator. We're going to implement the print. And then we're going to implement converting to a float.

All right, so the first thing that we should probably add is the str method because then it will help us in debugging when we print an object of type fraction. So let's define __str__-- again, no parameters except for self because that's the object we're calling this method on.

And again, however you want to form this string is up to you. You can use f-strings or a variable that you keep adding on to. I just did it straight in here with concatenation. So I've got the numerator slash the denominator as a very reasonable way to represent a string-- so 3/4 as 3 over 4. OK. So one thing, I guess, to keep track of is if you're concatenating, you just have to remember to cast the strings, if it's a number or something that's not a string.

So let's try it out. I've got three fraction objects here. So the first two we've already seen. So I've got a fraction representing 3 over 4, a fraction representing 1 over 4. And f3 is now going to be a fraction representing 5 over 1. If we print f1, again, Python asks, hey did you implement an str method in your class definition? Yes, you did. Good job. Let me use it.

So then it uses this. So it grabs the numerator of f1 slash the numerator of-- sorry, the denominator of f1. So this will print 3, the numerator of f1, slash the denominator of f1. Same with f2, except that now it's going to grab f2's numerator and denominator, 1/4.

So notice, now it's not doing the divisions like it did before-- or sorry, never mind. We're not there yet. There's nothing to divide. It's just grabbing the numerators and denominators and just printing them out. It's not doing any divisions. Now when we print 5-- the fraction object representing 5 over 1, it prints 5/1. I don't like that because it looks weird. Do you like that? No.

So then I'm going to have you fix it. Change the str method just a little bit such that if the denominator is 1, just have it print the numerator. And otherwise, the representation should be as before, numerator slash denominator. So this should be down line 140-- where is it? Very far down, 265.

OK. Anyone have some code for me? Yes?

STUDENT: [INAUDIBLE].

ANA BELL: Yep.

STUDENT: And then do an else statement and put the figures that we had underneath the else?

ANA BELL: Yep, we can do an else. That else is not needed, I don't think, because if we dropped into the if, we just immediately return. Otherwise, we would just do the remaining thing, but perfectly fine. Yeah, and let's run it. So a is a fraction representing 1 over 4. So it nicely printed 1 over 4. And b, the fraction 3 over 1 is just printing 3. Good. Questions about this code?

OK. For the remaining lecture though, we're not going to use this modified, this nicer, better str method. So let's just forget what we just did and just remember that it looks like this. OK. So now, let's implement the dunder methods for addition, multiplication, division, things like that.

So I'm going to do the multiplication just because it's not as long for the numerator, so just convenience factor here. The left-hand side, I've got our old simple fraction code. And the right-hand side has my new fraction code. So the old SimpleFraction code, remember, had this times method that took in self and oth and calculated a new numerator, a new denominator, and returned this.

Now, my new fraction code will no longer need to call times. So we're not even going to implement a method called times. Instead, we're going to implement the method behind the scenes for the shorthand notation star to multiply two fraction objects.

So we need to implement def __mul__. Parameter list is the same because we still have a thing before the star and a thing after the star as the two fraction objects we'd like to multiply. Within the code itself, the calculations of the new numerator and the denominator are the same as well. We're still grabbing the numerators of self and other, the denominators of self and other, and multiplying those together. What's different is in the returns, right? What was the return type for the times method?

STUDENT: [INAUDIBLE].

ANA BELL: A decimal. Yeah, a float, exactly. What's the return type of my new method, the mul?

STUDENT: A fraction?

ANA BELL: A fraction, exactly. So yes, I am operating with fraction objects. So I'm expecting that the return type of this method, the star, __mul__, is also a fraction object. So then I can just keep working with fraction objects throughout my code, not having to worry about whether this thing is now a float or not.

All right, so how are we creating this fraction object? Well, just like we would create a regular fraction object up in the previous slides, so here. Here's an example of us creating a new fraction object-- numerator 1, denominator 4. Well, same here, this method will return a new fraction object whose numerator is the thing that I just calculated, the top. And the denominator is the thing that I just calculated, bottom. Does that make sense? OK.

So this one returns a float. This one returns a fraction. Let's run it. So a is fraction 1 over 4. b is a fraction representing 3 over 4. Good, those are the numbers we've been working with. If we print a, the print statement says, this is the fraction object 1/4, whose representation is 1/4.

Now, if I use the star operator between a and b, the thing before the star is kind of like the thing before the dot. It's the self. It gets mapped to self in my __mul__. And the thing after the star, the second parameter so to speak, is the one parameter that my method takes, other.

So this will run the mul method behind the scenes. So Python, when it sees that star, asks, do you have a mul method implemented in your class fraction? Because the thing before the star is a fraction object. Yes, we do. What does it return? Well, it does the multiplication. And in the end, the return of this method is this thing here.

So I literally just made this-- I just copied this from the return using the numbers of a and b. So it creates a new fraction object whose numerator is 3 and denominator is 16. So c equals fraction parentheses 3, 16, basically just another fraction object.

So now, when I print c, It's going to use the str method for a fraction object because c is a fraction object right there. That's exactly what c was. So this will also print the way we asked to print fraction objects, numerator slash denominator, 3/16. Everyone OK so far?

OK. So the following lines are all equivalent. Using the shorthand notation-- very nice, very Pythonic way to multiply two fractions together. But behind the scenes, this is just running a method. So of course, if you really want to, you can just use the same old way of calling a method-- thing before the dot, dot, method name, parentheses, parameter list.

So here, thing before the dot is a dot the name of my method, __mul__, parentheses, all of the parameter list except for the thing I'm calling it on. So those two are equivalent. And of course, last time, I mentioned a way to hopefully demystify running these methods where the self becomes this thing before the dot.

You could call the method on the name of your data type, the type that you're currently creating, fraction. So fraction is not an instance. a was an instance. It was an actual object that we created. But fraction is just the name of my class. So if you call the method on the name of your class, then Python expects the full parameter list-- so something for self, something for other, something for whatever parameters you have.

And so there, we would explicitly pass in both a and b as part of my parameter list because the thing before the dot is not an object. So it doesn't map it to self. But I would never ever, ever run a line of code like this, this last one here. This is just for your information. It's non-Pythonic. It's just very verbose.

And so these dunder methods help us abstract away a bunch of these details. So how annoying would it be to always use dot notation when we want to multiply an integer with another integer? Can you imagine constantly writing 3.__mul__(4)? That would be very bad code. It would take forever to read.

And so we're abstracting away all the details for calling these methods into these nice little shorthand notations. And as I said, these shorthand notations exist for a lot of different operations. We saw print. You can do length comparisons, like equality, even indexing into things. You can always abstract all of those away into shorthand notations. And behind the scenes, these methods will be run.

OK. So big idea, exactly what I said, all these special operations that we've been using already, behind the scenes, these methods get run. And these methods were written inside the class definition for the types that we've been using. So when we index into a list, l square brackets 3 or whatever, there's a method being called behind the scenes in the list class to grab the element at index 3. I don't remember the dunder method name for that, but probably like __index__. I don't know. But there is some method behind the scenes. All right, let's do a couple more things. Sorry?

STUDENT: If we forget the dunder name for something, is there a way to ask Python?

ANA BELL: You can't ask Python, but you can look at the documentation. I think it's in python.org. There's a website that lists basically everything that's a dunder method, yeah, under categories, like all the indexing type stuff, all the numerical type stuff, yeah.

OK. So let's do one more thing. Let's say that-- well, we're working with fraction objects. And so the dunder methods that we're writing are now returning other fraction objects. So let's allow the user the opportunity to cast one of these fraction objects to a float, just in case they would like to grab the float value of 3/16. That's a very reasonable thing they might want to do. So let's get ahead of them and add that as part of our class definition.

So to cast things to a float, in this particular case, the dunder method for that is __float__. And all it's going to do is grab the numerator of self and divide it by the denominator of self. So this will just do a division. self.num is a number. self.denom is another number. It does the division. And this returns a float.

So here, when we multiply c is equal to a times b, remember that c became a fraction object with numerator 3 and denominator 16. Do you remember that? So then when we cast it to a float, down here, Python says, hey, did you implement the dunder method __float__? Oh yeah, you did. Let me just go ahead and do the thing that you want me to do inside it. So it takes the 3, divides it by the 16, and it prints 0.1875.

OK. Let's try it out a little bit more. So here, I've got 2 fraction objects-- one representing 1 over 4, the other one representing 2 over 3. I multiply those two together. Again, this gives me a fraction object because it's running the mul dunder method. And the mul dunder method gives me a fraction object with a new numerator and denominator.

So when I print the return of that, when I print c, this prints the new numerator, which is 2 times 1, divided by the new denominator, which is 4 times 3. So it prints 2 over 12. Does that look OK to you? I mean, it looks OK. But suppose you're doing calculations with a whole bunch of numbers. And at some point, you get two really big numerators and really big denominators.

But then you stare at it long enough and realize that big numerator divided by that big denominator is actually something like 1 over 4. So this is not reduced, which is fine. Our code is not doing the reduction. But it would be nice to write a method that allows the user to reduce a fraction. That would be really nice. So can we fix this? Yes, we can. Otherwise, we wouldn't be here.

So let's write this method to reduce a fraction object. It looks like a lot, but it's not, trust me. It's just a bunch of if/else's. So the first part of it is a little helper function, not a method. Notice, there's no self going on in this gcd function. This is just a regular function that I will use to help me get the greatest common divisor for the two parameters, n and d.

Because when I have two numbers, if I want to reduce them, I find the greatest common divisors. And I'm going to divide the top and the bottom by that divisor. And that will help me reduce it. So this gcd function helps me find this greatest common divisor.

All right, so here, I'm just defining the function. I'm not actually using it. And then I've got an if and an elif. So if the denominator is 0, something's super weird. So I'm just going to return none because having a fraction where the denominator is 0, maybe something went wrong.

Else, if the denominator is 1, we don't need to do any reduction. No reducing is needed. So we just return the numerator. And else, I do have two actual numbers that I maybe could potentially reduce. So let's just reduce them. The first line here runs this function, this helper function that I wrote, on the numerator and denominator to grab the greatest common divisor. So if it's 2 over 12, it'll find 2.

Then the next line here takes the numerator and divides by that greatest common divisor and casts it to an int because I want my numerator to be an int and my denominator to be an int. So it'll take the numerator and divide by, for example, 2.

Same with denominator, I'm going to take my denominator and divide by that same gcd I found, casting to an int. So my new top and my new bottom will now be used to create a new fraction object that is in reduced form, so 1/6 for the example 2/12.

All right, so here it is. This is my previous example, where I multiplied that thing that gave me 2/12. And then if I do c.reduce, Python will call the reduce method on c, so the object whose numerator is 2 and denominator is 12. And then this will reduce it to 1 over 6. And it will print, call the printstr method on an object of type fraction to give me 1 over 6. Everyone OK? Yes?

STUDENT: [INAUDIBLE].

ANA BELL: You could put it outside the reduce. But since it's being used specifically in the reduce, we'd like to just keep it within. If it doesn't need to be used by other things, we'll just keep it only to the scope where it needs to exist. But it can be outside, yeah.

OK. So one thing is weird here, though, right? This elif here. What is the type that gets returned from the else? You guys tell me. What's this type here that gets returned down in the else? Fraction. What is the type being returned in the elif?

STUDENT: Is it an int?

ANA BELL: Yes, an int. So if the denominator happens to be a 1, this method, reduce, returns an integer. If it's not, it returns a fraction. So if at some point in the future you're mixing-- you happen to reduce something that has a denominator of 1, you're now working with integers. And then potentially, you'd be doing further operations by mixing that with fraction objects.

So as an example here, I've got a fraction object-- a, 4 over 1; b, 3 over 9. Reducing a gives me a 4. Fine, it's the integer 4. And reducing b gives me 1 over 3. It's a fraction, 1 over 3. So the type, just to show you exactly that I'm not lying-- the type of the a reduced is an int. That's what the code is doing. And the type of b reduced is a fraction.

So then when we do the star operator between ar and br, Python is going to say, you're trying to multiply an int with a fraction. Did you ever tell me how to do that? No, right? We told it how to multiply a fraction and another fraction, but not an int with a fraction. And so Python will fail here.

So one thing that you can do to fix it is to change this elif here. So let's have everything consistent. So I want you to do this change. Instead of returning self.num, return for me a fraction object representing the numerator. All right, does anyone know? Just a small change. Instead of returning self.num, what should I return? How do I make a fraction object? Just invoke the name of a fraction, right? What's the numerator of this fraction object supposed to be? It's already there, self.num. What's the denominator of this fraction object?

STUDENT: [INAUDIBLE].

ANA BELL: Yeah, exactly. So it just returns a fraction object whose numerator is self.num and denominator is 1, exactly. All right, so now, all the different cases except for this randomly weird denominator being 0-- in case that happens, something's gone wrong, maybe-- all the other cases are returning a fraction object, which is good because now it's consistent.

STUDENT: [INAUDIBLE]?

ANA BELL: Oh yeah, exactly. So we did say we didn't want it to be 5 over 1. But this is actually using the old str method, where we didn't do that check. So it will print 5 over 1. But if we do the check if self.denom == 1 then return str self.num-- if we add that, then it won't do that, yeah. But this is just using the old method that doesn't do that nice formatting for us. Questions? All right, we've been working a lot with returning new objects of the same type that we're writing. That's a new thing today.

OK. So what's the purpose of these two lectures? So hopefully, it shows that it's very useful to bundle data and behaviors together. So the ultimate goal when we're writing programs is to write code that's modular and organized because, in the future, you might want to build upon this code.

In the future, you might want to read this code to use it for something else. In the future, other people might want to read this code or use this code, this class that you wrote, to build more complex classes, like we used the coordinate class to build a circle class.

Other people might use your circle class to build-- I don't know-- a sphere class or something like that, something more complicated. And so it's really nice to create these little data types that are organized, modular. And so we're basically bundling together these data-- so what makes up your object-- and behaviors together, so we can use these objects in a nicely consistent way.

So remember, back when we were learning about functions, the ideas of decomposition and abstraction were very important. Functions basically took a chunk of code and decomposed them into one module that we could reuse many, many different places. And we abstracted away the details of the function through docstrings, so people didn't have to slog through a whole bunch of code to figure out what the function did. All they did was read the docstring and they knew exactly what we wanted.

Now, object-oriented programming in Python classes have that same big decomposition and abstraction energy, right? They've got a bunch of modules that we're creating here, where we're bundling together data and behaviors, so we can create a whole bunch of objects that behave in the exact same way-- nicely consistent-- so that we know that if I create a coordinate object here, it's going to have an x and a y value. And if I create another coordinate object, it's also going to have an x and a y value. It's not suddenly going to have an x, y, and z value.

So creating these objects that work in a consistent way is a very-- decomposition and abstraction are working with the ideas of decomposition abstraction just like functions did. OK. So next lecture, we will be starting on-- we'll do a little bit more of these classes. And then we'll start on inheritance, so having parents be objects that we created. All right.