[SQUEAKING]

[RUSTLING]

[CLICKING]

**ANA BELL:** OK, so let's get started. Today's lecture will be the last one we have on object oriented programming and creating our own data types with Python classes. So in today's lecture, we're going to go through an example that's more involved. We're going to be creating our own fitness tracker object. And specifically, we're going to create a class that implements the idea of a workout.

And the slides for today, are going to feel very similar to the slides from Monday's lecture. A lot of them are just kind of reinforcing the same ideas we saw last lecture on creating getters and setters, creating class variables, and the idea of inheritance. But we're just going to do it in the context of this more involved example, the fitness tracker, OK.

So let me remind you, first of all, something we've been talking about and hopefully, you understand at this point in our lectures on object oriented programming, and that's the idea that when we write our own object types, we're writing code from two different perspectives.

The first perspective is the one on the left-hand side here, where we are making design decisions for how we want to implement this new object, this new data type. And when we make these design decisions, we decide the name of the object. We decide the attributes, which are either data or procedures that we want the object to have.

And then once we've decided on this, and we've implemented our data type, then we can start to use the data type. And to use it, we are creating a whole bunch of objects of this type. And we're manipulating these objects in some interesting and useful way.

So the left-hand side, we're creating this blueprint, this abstract notion of an object. And the right-hand side, we are creating actual instances that we manipulate. So up this object oriented set of lectures, we've really just been working with the right-hand side. We've been working with objects that other people have created, but the big idea of these set of lectures is that we now get to create our own object types, so we get to write this code here.

OK, so we're going to write code to create a tracker for workouts. And today's lecture, there's going to be a sequence of things that we're going to do. We're going to start out with a really simple workout object. And then we're going to improve on it. So I've actually set out a little roadmap here on the board that we can follow. So every time we finish sort of a little section, we'll check it off so just easier to understand where we are in today's lecture.

So we're going to create, first, just a very simple workout object in the same spirit that we've been creating objects. Then we're going to improve on it a little bit by adding nicer methods and things like that. And then we're going to go through the idea of inheritance to create very specific types of workouts.

So if we think about workouts, we have many different kinds of workouts, right. We've got biking, swimming, running, but really, at the core of all these workouts, if we think about the information related to just a very generic workout, not a running or swimming specific one or a biking one, just a generic workout, there are some common properties that all of these workouts have. So I've listed them here.

A workout might have an icon associated with it. So this or this or this, but whatever it may be, there is an icon property for a workout. The kind of workout so biking, swimming, running, things like that. A date, so maybe a start date and an end date, start time, end time, that kind of information. The heart rate, maybe average heart rate throughout that entire activity, the distance and the number of calories burned. All of these properties are common to any kind of workout that we might want to create.

But now that we have specific kinds of workouts that we might want to create, we can actually think well, in addition to these common properties, a swimming specific workout might actually have some more information that we'd like to save, and we'd like to allow the user to work with so the swimming pace, maybe the stroke type, the 100-yard splits, things like that for swimming. And for running, we might want to show the user the cadence, the running pace, the mile splits, and maybe the total elevation throughout that run.

But the idea here is that we have some core set of properties, that no matter what kind of workout we're creating, we would like to save, and we would like to allow the user to store and to do operations with. Now when we implement our workout class, we're not going to implement all of these. They're not all necessary. We're going to just keep some of the core ones.

So the ones we're actually going to implement in this class are italicized. So we're going to keep the icon and the kind of workout, the start time, end times, and then the number of calories burned. That's something that we're going to just save as the common set of attributes for a workout. But of course, you can see that if you make the design decision to improve upon this workout class, you might include a bunch of these other ones, as well.

OK, so we're going to have to decide the data attributes. So we just mentioned on the previous slide, when we make design decisions, we figure out the attributes that we'd like to have for our object type. So that's the data and the behaviors. For the data for our workout, we've decided it would be the start time, the end time, and the number of calories burned. So those three things together maybe start time is a string, end time is a string, and calories is a number, either a float or int or whatever. Those three things together make up the object a workout.

And then in terms of functional attributes, so these are the methods that our object might have, well, we can have, of course, the ability to tell us the number of calories burned, so something like a getter method to set the number of calories burned if we accidentally inputted the wrong number, reset it. And then maybe something like displaying an information card. So something like this if the user asks us to print a workout object, we might display information in this nice manner here.

All right, so let's start defining our class. So this is a very simple workout class. So we're going to do the box number one up there before we improve on it. So this is in the same spirit as we have been defining classes in the past three lectures, all right. So we tell Python we're creating a new object by saying this is a class and the name of this object. So the type of this object is Workout, and the parent of this object is the generic Python object, so far so good, right.

Now, the first method we have to implement is the init method. It tells Python how to create an object of type Workout, the constructor. And we've got a bunch of parameters in here because it's just a regular function that's a little bit special. The first parameter of every method is, of course, self.

Because when we call a method, we call it on an object. So some object dot this method name. The thing before the dot, effectively, gets mapped to the variable self, which is why every one of these methods has self as the first parameter. And then we've got a whole bunch of other parameters for how we would like to initialize the Workout object.

So we're going to say, when we create a Workout object, we're going to tell it the start time, the end time, and the number of calories burned. So that's the function stub, I guess. That's how you create the object. And then what does this function actually do? What does this method actually do? Well, it does some of the usual things that we know at this point. So it basically maps every one of these input parameters to data attributes self.start, self.end, and self.calories, OK.

But in addition to just saving these as data attributes, the things that are passed in when you create the object, we would like to do two more things. We're creating two more data attributes. So in total, a workout object is defined by five data attributes. These last two data attributes, we don't need to pass in.

We're just going to, by default, make them two strings. The icon is going to be the string, this sweating person emoji. And you can have emojis inside strings, which is actually pretty cool. And the kind is going to be just the kind of workout, so we're just going to set it to be the string workout. When we create running workouts, it'll just be the string running. When we create biking workouts, it'll be a string biking. Whatever you want it to be. And we're going to see where these show up later on.

OK, so that's the definition of my class Workout, and then, for now, that's it. That's all we have in terms of the class definition. Now what happens when we create an actual Workout object? Well, we invoke the name of the class. So we say here, Workout. And we're going to save this object, to the right hand side of the equal sign, as variable my_workout, so far review.

We pass in the parameters that the Workout object expects. So here's a string representing the start time, a string representing the end time, and the calories burned for this particular workout, 200. Yes, good.

OK, so then we can add a whole bunch more methods to our class. That was just the init method. But last lecture, I mentioned that it's important to add getters and setter methods to allow the user to grab or set various data attributes. So here, I've got three getter methods to grab for me the calories, start time, and end time, and three setter methods to set the calories, start time, and end time.

All right, so what I wanted to show you, and this is not something we've actually seen before, I wanted to show you that every time you create an object of some type or even an object that already exists, you can actually look into where this object is stored in memory, which is kind of cool.

So if we think about the class definition that we've done so far. So not creating an actual instance, just defining the class. This class definition is actually kind of like an object stored away in Python memory. So here I have my Workout class. And associated with my Workout class definition, Python knows about all of these methods that you're allowed to do with this Python class.

And this is called the class state dictionary. So it's a dictionary that basically holds the state of your object. So I wanted to show you what that looks like in code. So this is my Workout class. And the way you access the state dictionary is by invoking the name of your class. So not an instance, the name of the actual class dot this Dunder method double underscore dict double underscore. So this holds the state dictionary.

And if we just access the keys, we're going to see here every single method we've defined in our class. So you see here, here's my dictionary. I could cast it to a list if I wanted to but not necessary. But you can see every single method that we've defined. So all our getters, all our setters, the init method, the double underscore doc actually grabs for us this docstring that you've put right under the class definition. So that's kind of cool.

So that's the dictionary keys, and of course, as we know, keys have values associated with them. So the values associated with each one of these keys is going to be, and we can see here, so for example, the value associated with the docstring, is going to be, literally, the thing that I printed out-- the little docstring that I put right underneath my class definition.

So now it knows the docstring for this class that I just created. And the values associated with my getter methods and my setter methods and the init method and all the methods that I created, are just the locations in memory where Python can find these methods to run. They don't have actual values associated with them, of course, because they're just method definitions, but Python just knows where to go in which memory location to actually run this function. OK, so that's kind of cool to know.

OK, so that's the state dictionary of my definition, the implementation of my class. Now, what happens when I create an actual instance? So here I've got my_workout equals, and now I've got this actual instance of this class type Workout.

When Python sees this line, it says, OK, what kind of object do you want to create? A workout object. It looks at the init method of that Workout object, and then it runs all the lines associated with that Workout object. So now it creates a new object in memory puts that at some memory location. This object is going to be of type Workout class.

And now this object is going to have its own state dictionary, in the object state, dictionary we're not storing methods or things like that. We're storing the actual data attributes associated with this object, all right. So this object, all the data attributes are all the things that you access via the self dot keyword, self dot icon, self dot kind, self dot start, self dot end, and self dot calories.

So we can actually go in the code, just like we did when we looked at the class state dictionary, and look at the state dictionary for one specific object, one instance. So again, we can call the double underscore dict method on this instance. So now I have an actual object that has some values associated with it.

And if I look at just the keys, we see these are the data attributes associated with an object of type Workout. I've got my five data attributes. And then the values associated with those keys, are going to be the values that are specific to this object. So my start is this date here, my end is this date here, calories was 200. The icon was the little sweaty person emoji, and the kind of workout is workout. So it's kind of neat to be able to look into that sort of detail of where things are stored inside memory.

OK, so we saw how to create an instance of an object, and we can create a whole bunch of workouts that we then store. And then we can use dot notation to access all of these attributes. So we can either access attributes directly, or we can access methods. We already know this.

So last lecture, I said that you can use dot notation to access data attributes. So here we're accessing the calories value. And that's fine, but what's preferred is to use the getter methods. so get_calories will, in this particular case, return the exact same value as just accessing the calories data attribute.

But the note that I made last lecture, was that it was better to use a getter method because the implementation behind the scenes might change. And if the implementation changes, then if you access the calories method directly, or sorry, the calories data attribute directly, your code might crash.

But not only that, somebody who's writing a getter method for this workout function, might actually make that method be a lot more complex than just returning that data attribute. And that's what we're going to see in the next slide.

So the idea behind using these consistent methods, instead of accessing using data attributes, is that you want to keep information hidden. You don't want to start messing around with looking at how something is implemented. Because that goes against the principle of abstraction, modularity, and information hiding. You want to keep things hidden because you want to use the objects that somebody else has created in a nicely consistent manner.

The way we use them in a consistent manner is by always using methods that are associated with that object type. And so using getter methods might have seemed inconsequential when we wrote the Animal class last lecture, but it's going to be a lot more important in this particular lecture.

So with that, we've finished our simple Workout class. And now we're going to change the implementation just a little bit. And what we're going to do is, we're going to make a change to the way that we store the information. We're going to use a class variable. And I'll remind you what a class variable is in the next slide, and we're going to make a change to the get_calories method.

And we're going to allow the user to say, hey, I'm going to create this Workout object. But I don't know about you, I don't know how many calories I burn when I do a workout for 40 minutes. I don't know that right off the bat.

So if the user doesn't supply the number of calories burned, we're going to have our get_calories method estimate those calories burned based on the duration of that workout. So we're going to allow the user to either supply the number of calories, in which case they probably know what they're doing, and then when they ask us to tell us to get the calories, we're going to use those.

Or we're going to allow the user to not supply the number of calories, and instead estimate those calories based on the duration that they said this workout lasted. All right, so that's the big change that we're going to do here in the Workout class. So we're going to do a better get_calories method.

All right, so this is the new implementation of my Workout class. First thing you'll notice, is we're using this class variable. We talked about this last lecture when we did the rabbits example. In the rabbits example, we had each rabbit change this class variable value.

In this example, I'm not going to change this class variable value. I'm actually just going to use it as a variable that every one of these instances is going to be able to access. And I'm just not going to change it, which is fine. You don't have to change this class variable.

So this class variable will represent how many calories per hour are burned, so just a number. And then the init method, and again, we're going to make a different init method than what we saw in the previous slides, the init method is going to be new and improved. We're going to take in still the same number of parameters, but the calories are going to have a default value.

So if the user actually passes in the number of calories, the value for calories here, self dot calories will be whatever the user passed in. But if the user doesn't pass in the number of calories, then this parameter here, self dot calories will be none. None being used to represent the absence of a value. So two options here when we create the object.

Other things you might notice is that the self dot start, so the start time and the end time are no longer just start and end. I'm going to talk about this on the next slide. But essentially what I'm doing here, is I'm saying the start and end will be passed in as strings, just like we have been in the past, like, September 1, 2022 1:35 PM. That's fine. We can still pass in the start time as a string.

But when I'm storing it inside my object, I'm actually going to store it as whatever this thing gives me. And this thing is actually going to be returning or parsing the string as a new data type, something we've not worked with before, called a datetime object. We're going to look at this on the next slide in a little bit more detail. But for now, all we need to know is that the self dot start and self dot end will be a new data type, a datetime object.

So that's my init method, so few changes. Now my get_calories method will look a little bit different, as well. We're not just returning self dot calories like we had in that simple Workout class. We're going to do a little switch.

So if the user supplies the number of calories, so if the calories here were actually passed in, then we don't resort to the calories being none. Calories will be 100 or 450 or whatever it is. And then this if statement is false, so we go in the else, and we just return that value. So it's exactly the same behavior as in my simple Workout class from back there.

But if the user does not supply the number of calories when they create an object, then the calories will be none here. When I create my object, the data attribute self dot calories will be none here. So when I ask the workout to tell me how many calories I burned, we're going to go inside the if statement. And we're going to do something.

The thing we're going to do is subtract the end time minus the start time. And something like this is allowed on a datetime object but obviously not allowed on strings, which is why I'm converting these strings representing a date and a time into this datetime object. This subtraction here gives me something that's called a timedelta object.

And it's just a new type of object we haven't really worked with before, but it's an object type that we can run a method on. And the method is going to be the total_seconds. So for this timedelta object, so 10 minutes or 18 minutes or whatever it may be, if we run this method called total_seconds, it will tell us how many seconds are in that timedelta object. Divide by 3,600 to tell us the number of hours, and then multiply by the class variable cal_per_hour will tell us how many calories were burned in that elapsed time. Yeah.

**AUDIENCE:** If we can do like workout dot and then all of that like--

**ANA BELL:** Oh, workout dot is just this thing here. Workout dot cal_per_hour, that's just this, and then we multiply by that number. Questions about that? OK, so essentially this is going to do the estimation for us for how many calories we burned in some number of hours or some number of minutes.

Now, let's demystify this start and end time stuff. So the way that we are converting this string to a datetime object, is by using this library up here. So a library is a collection of objects, a collection of maybe also functions, that all deal with the same type of thing. So in this particular case, they all deal with dates and times and manipulating dates and times and things like that. In the last lecture, we saw an example of a library, that random library that allowed us to do operations with random numbers. So it's just a nice collection of functions and objects that deal with one common thing.

So in this particular case, there is a function inside that library, this parser dot parse function that takes in a string and can parse it to this datetime object. So if we print the type of start date and the type of end date, it will show us that it's this type datetime thing. So it's a new object type we haven't worked with yet, but it's an object type like a list is, like a dictionary is, like our workout is.

And so the reason why we're doing the conversion is because we don't want to deal with the messy part of grabbing in a string and then figuring out how long the elapsed time is based on just parsing characters throughout the string. I certainly don't want to do that, but you know what? Somebody who is passionate about doing that did it in this nice little library for us.

So all we're doing is just reusing the work that they've done to save it as this object, and then, they basically said let me implement the minus sign to work with objects of type datetime, and it makes things like this very easy. We can just subtract two dates from each other, and it will tell us the elapsed time. We can run a method on that elapsed time to tell us how many seconds that elapsed time is, so pretty cool. Yeah.

**AUDIENCE:** Does those total seconds get imported for the parser?

**ANA BELL:** Yeah, the total seconds gets imported with the date util parser thing, yeah, exactly. It's an operation that can be run on this date timedelta, I think, type object here.

**AUDIENCE:** So it's like total hours, total minutes.

**ANA BELL:** I think there might be total minutes and total hours also, yeah, yeah.

**AUDIENCE:** So this parser class, and then dot parser dot total_seconds are--

**ANA BELL:** Yeah, exactly, yeah, exactly. Yeah, so yeah.

**AUDIENCE:** Should the code should be imported thing?

**ANA BELL:** Yeah, so we usually import all our stuff right at the top. So I was just going to show the code. So here I've got everything that I need to import way at the beginning. So it's kind of like Python goes and copies and pastes everything in those files and puts them in your file. So now everything that's defined in those files, is now accessible in your file. You just have to do the dot notation on these libraries here.

So I just wanted to show you down here. So here, I shouldn't have imported again, but it's just part of this exercise. So here I've got the parser being imported. I've got the start time. These are just strings, nothing specia about them. And we can parse them. So I've got these strings parsed.

And the types of these objects, again, are not strings anymore now that I've parsed them. start_date and end_date are now these datetime objects, datetime dot datetime. And then we can do operations like this. So if I just simply subtract one time from the other and print that timedelta object, Python puts it in this nice little format for me. I should just comment these out. It's hard to see. It puts it in this nice little format for me.

So here's number of hours colon number of minutes colon number of seconds. So this is the str method that was implemented for that kind of object. It prints it in this nice little form hours colon minutes colon seconds. And then we can do this useful thing, which is what we're doing in our code, we can run the total_seconds function on an object of this timedelta, and it tells us that this 10 minutes is equivalent to 600 seconds. So very, very cool, very useful. And we don't even need to know how any of that is implemented. We just make use of these functions.

What's cool about the parser, though, and this will be really, really cool, you can actually write the time and the date in any format. It doesn't have to be month slash day slash year space this. So this is kind of how I wrote this one. We can actually write it something like Sept 30, 2021 like that. And it knows how to read that.

Or we can write out September all the way put the comma there, put the comma there, put the pm lowercase and closer to the time, and it knows how to read that as well. So it knows how to parse all these different ways of writing the dates and save them as these datetime objects for using in this very nice very readable way. Isn't that cool? OK, so very useful if you ever want to work with date types.

So now this is our state dictionary so for how we ended up with our simple Workout class. But what are the changes we made to improve it? Well, in my state dictionary, I added my class variable calories_per_hour. So now this calories_per_hour is available for any instance that I create. We already knew this, but this is kind of a representation of that. And we didn't add anything to the instances. Those haven't changed.

So little aside on class variables, so this cal_per_hour here is available for all of these instances. Now, a class variable is just like an instance variable. We can access it from within the class definition, which is how it should always be done. But Python being Python, they allow you to access that class variable from outside the class definition, as well.

So we can do something like this. So we can call the cal_per_hour class variable on the name of this class outside of the definition. This is my class definition. It ended here. And this is just code that's outside the definition, and Python will be happy to tell you what that value is.

Python will also be happy to tell you what that value is if you access it through an instance. So here I've created an actual instance of type Workout, so I'm not calling the cal_per_hour on the name of my class, I'm just grabbing it through one of my instances. And if I print instance dot cal_per_hour, Python will also happily tell me what that value is.

And Python being Python, they're going to allow you to change the value of that cal_per_hour, outside of the class definition, as well. So here, outside the class definition, I'm going to say workout dot cal_per_hour = 250. So now, the cal_per_hour is changed permanently to 250 no matter how I access it, either by calling the name of my class directly or by calling the class variable through one of the instances.

So, no good, never ever work with these access, data attributes or access class variables outside the class definition. If you really want the user to be able to do something like this, then write a method for it. And then they can change it or access it in a consistent way, the way that you want them to access it.

OK, so just a little bit of practice for you guys to create a couple of Workout objects, just to make sure everyone's on the same page we understand what a workout object is. So just create for me two objects and then print the calories for these Workout objects.

So the first one I would like you to create, name the variable w_one. This workout started in January 1, 2001 3:30, and it went till 4:00 PM. And you want to estimate the calories from this workout. You don't know how many calories you burned, and then print the value for that calories. And then the second object, same start date same end date, but you know that you burned 300 calories. So create these two objects, and then print the number of calories burned.

So this is online 199. And it's OK to scroll back up to the init method of Workout, just to see how it's implemented. No reason you should have memorized it by now.

All right, how do we create these two objects? What's w_one equal to? Yeah.

**AUDIENCE:** [INAUDIBLE]

**ANA BELL:** Yep.

**AUDIENCE:** Is the date first?

**ANA BELL:** Yep, the start_date would be first. So I can just--

**AUDIENCE:** [INAUDIBLE] put that as a string.

**ANA BELL:** As a string, perfect, yep.

**AUDIENCE:** [INAUDIBLE]

**ANA BELL:** Yep. I'm not saying. I don't think I'll say at. I don't know if that works. And then the end_date right is the next one. So this one's 4:00 PM, right like that. We can do that. And then what else do I put? Or do I put anything else?

**AUDIENCE:** If you want [INAUDIBLE] calories [INAUDIBLE].

**ANA BELL:** Exactly, yeah, in this particular case, I'm going to let it default to none. And then how do I grab the number of calories burned for this object, for this Workout object? How do I print that out?

**AUDIENCE:** Don't you want that [INAUDIBLE]?

**ANA BELL:** Yep, so I just call the get_calories method on w_one, and let's slap a print statement around that, like that. Yep.

**AUDIENCE:** Mine, when I do that-- oh, wait. let me do that.

**ANA BELL:** Perfect. So what is it? A 30-minute workout at 200 calories per hour, it's 100 calories burned. Second one will be pretty similar. So I'm just going to copy and paste. What's the difference between this one and the previous one? When I create my object, what's the one difference? Yeah.

**AUDIENCE:** Don't you pass in the number of calories [INAUDIBLE]?

**ANA BELL:** Yep, exactly, we will pass in 300 as the last param here. And so then if we run that, 100 was my first print statement and 300 is my second one. So it relies on the number that was passed in as opposed to calculating it by estimating it based on the start and end.

All right, everyone OK with this? We all understand get_calories method, perfect. So we've finished our improved method here. We saw this better get_calories method, very neat method that allows estimation. And we saw a little bit about using these datetime objects.

OK, so the next stop, the rest of this lecture will be implementing one subclass of this state of this workout object called a RunWorkout class. And so we're going to use the idea of hierarchies and class inheritance to do this. So let's remember a little bit about hierarchies In terms of Python.

So when we create a class that we know will be this parent class, that's a base class. It's the most basic thing that we'd like to work off of. We call that the parent class or the superclass, and this one parent class, can have many subclasses associated with it.

So in this particular case, just as an example, we can have two types of workouts, one outdoor workout and one kind is an outdoor workout, and the other kind is an indoor workout. And both of these are Workouts. So everything that a Workout has and everything that we can do with a Workout, we will exist in outdoor workout and indoor workout, and we'll be able to do with outdoor workout and indoor workout.

So a child class is a parent class. A subclass is a super class. But these subclasses can bring about some of their own quote unquote "ideas," their own attributes. So for an outdoor workout, we might add more information, so add more attributes, maybe location, something like that. For indoor workout, you might not need a location. You might not add any extra data attributes.

We might add more behaviors. So for outdoor workout, I don't know, you add some different behavior than just a regular workout. Same for indoor workout, or you might override behaviors. So you might change something that Workout does to be specific to an outdoor workout.

And of course, we can create as many of these subclasses as we'd like. So for outdoor workouts, we can now have two different types of outdoor workouts, running or a swimming. And for indoor workouts, we might have treadmill or weights types of workouts. And whenever you create these child classes, they inherit everything that their parents has. So a running class is an outdoor workout and by default, it's also going to be a Workout right because outdoor workout was a Workout.

So what we're going to do in this RunWorkout class is, I'm going to show you three methods implemented. The first one is going to be just reusing something that our parent can do. The second one is going to be overriding a method that our parent already can do to make it better and more specific to the child class. And the third one is to add a method that our parent didn't even have. So we're going to do these three things in the run workout object.

OK, so let's remember this example here about common properties that all of our Workouts have. So this is basically us implementing our Workout superclass. So I know we did implement all of these, but in theory, we can implement all of the things that are highlighted in yellow in our parent class. And these are common no matter what kinds of Workouts we create.

Now, in the Python file, I actually have a swimming subclass. We're not going to go over the swimming subclass in the lecture, but please feel free to go through it in the Python file for this lecture. And I think you'll also be working with it in recitation on Friday, as well.

In lecture, we're going to be creating a subclass that's specific to running. And this running class will, of course, inherit all of these properties that our parent workout class has, but we're also going to add our own data. And we're going to override some data that the workout class has and things like that.

So specifically, the only thing we're actually going to implement that's different than a regular Workout, is to add an elevation attribute. So beyond that start_time, end_time, calories an icon will also exist, and the kind of workout it will also exist. Those are our five data attributes from Workout. And we're going to add elevation for running workout to make six.

OK, so this was our parent class, just as a reminder, this is what it looked like. We had our class attribute here, class variable, and this init method here. The class Workout, their parent was the generic Python object. Now, when we create our RunWorkout, our parent will be the Workout class. So we don't just want it to be a Python object. We want it to be a Workout object.

So as soon as we do that inside our code, Python says, all right, let me just grab all of this stuff, everything that's defined inside your parent class, this Workout. And quote unquote "copy and paste it into this class." So right off the bat, we've got all of the things that Workout has.

But that doesn't quite work with our RunWorkout. Because when we create a RunWorkout, and again, this is a design decision, we would like the user to be able to pass in an extra parameter here, the elevation value. So in addition to the start_time, end_time, and the calories_burned, I'm going to slip in this elevation value right before the calories_burned parameter.

So when I initialize my RunWorkout, I could, theoretically, pass in four values string, string, number, number. Or since elevation has a default parameter, and calories has a default parameter, I could pass in just string string, and those other two will default.

So what is this init method doing? It's calling the super() function. I know we haven't done this before, but I just wanted to show you this. This is another way to ask Python to tell you who your parent is. So when you run this function super() inside a class definition, Python effectively returns-- so the replacement of this function, this is just a function it has a return value. The return will be the thing in the parentheses here.

So effectively, that line becomes workout dot double underscore init double underscore exactly what as we saw in the last lecture when we did animal, rabbits and all those done. All right, so what we're doing here, is we're saying, well, I know workout can do all those initializations for me, so let me just take advantage of that, not copy and paste it, and just let Workout do the job and initialize all that stuff for me.

So this line of code here, initializes the start end times right by parsing them, the calories and the icon to the sweaty person emoji, and the kind to workout string. But since this is a RunWorkout, I would like to replace the icon with the little running person emoji. And the kind of workout this is, is no longer just a workout. Let's say it's a running workout so it becomes a string "running." So I'm overriding those data attributes to be the strings.

And then the thing that RunWorkout has that Workout didn't have at all, is this elevation data attribute. So self dot elev is now a new data attribute that did not exist in the regular Workout.

And then I've got these two methods down here, a getter for the elevation and a setter for the elevation. Nothing fancy with them, they just return and set. Yeah.

**AUDIENCE:** Can we overwrite super init [INAUDIBLE]?

**ANA BELL:** Yeah, so the Super init calls the parent's init method. And the reason I do that is because I know the parent can just initialize all that stuff for me. So I'm just taking advantage of the fact that it does all that stuff for me. You can imagine the init method, maybe, it also checked the types to make sure that the person who created this to enforce that start is a string. End is a string. All that extra code that would happen in the init method of Workout, we would just let run with this line here. So we don't have to copy and paste it. Yes.

**AUDIENCE:** Do we also have to take by leaving the line above right in [INAUDIBLE]?

**ANA BELL:** Yes, you mean these two?

**AUDIENCE:** The [INAUDIBLE] so that [INAUDIBLE]--

**ANA BELL:** Yep.

**AUDIENCE:** --needs to have start and calories. And the Super needs to have at least start, end, and calories?

**ANA BELL:** Yes, exactly, actually, yes, you're right. So the reason this works is because the init method of my Workout takes in the start and the end and the calories. If I said that the init method of RunWorkout doesn't need start and end or something like that, I wouldn't be able to run this init method, exactly. Or maybe I would run it with some defaults or something like if you actually want to run it, you have to pass it-- you still have to follow the stub of that init method, right, yeah, that's a great point. yeah

**AUDIENCE:** [INAUDIBLE] you said, I feel like before passing the init of the parent into this one without having to write the Super thing.

**ANA BELL:** Yeah, so you're right. I wasn't writing the Super thing. I was just naming the parent directly. So in the animal one, I said, animal dot double underscore init double underscore. And in this particular case, I'm just showing you a different way to do it. Maybe you don't know who your parent is? In that case, you can just run this function, and it tells you who your parent is. But this line would work just as well if I said workout, write this thing, dot double underscore init double underscore. And that would be exactly the same as I had done last lecture.

So let's look at the state dictionary for this one, for this new addition here. So this is a state dictionary of just my plain old Workout class. We saw this before. It's all my getters, all my setters, the init, and this cal_per_hour from the new and improved Workout class.

Now, my subclass, the RunWorkout the super() method basically says, hey, you are a Workout. The super() method, the state dictionary for it will additionally have this getter and this setter, these two methods. We're not copying all this stuff all over again, down in the RunWorkout state dictionary because that already exists up here. But in addition, the RunWorkout has this get_elevation and set_elevation method. Those are the only things that we've defined in this class.

So then when I create a RunWorkout instance. So this is an instance of RunWorkout, not Workout. Python points to this RunWorkout class. And the data attributes for a RunWorkout instance are going to be, of course, all the data attributes of a regular Workout, those five, plus the elevation. The new data attribute that I just added.

So we're using inheritance in this particular case in the spirit of modularity, in the spirit of abstraction, in the spirit of writing code that's reusable, that's readable, that's understandable in the future. So if we were writing the RunWorkout by basically copying and pasting everything in there all over again, it would be a really long class, where most of it was just a copy and paste off of the Workout.

So now, if we define it in this way, we can easily see new functionality and new data attributes that RunWorkout has, in addition to just being a Workout object. So all those good things for writing very nice clear code

OK, so now we're going to look at a method that's being reused from our parent. And that's this double underscore str method. So this str method, it looks like a beast. It's very, very long, but I promise you, it's not so bad.

So this str method, let's remember what it does. It tells Python what to do when you print an object of type Workout because it's defined inside the Workout class here. So I'll show you what it actually looks like in the actual Workout class. So here's my Workout. There's my class variable, my init, my getters, my setters, all that, and then here's my str method. It's long, right? It takes an entire page.

This is not a method that I would like to copy and paste in every single one of my subclasses because that would be a lot of code. Again, against the spirit of abstraction, modularity, all that good stuff. So what we're doing is, we're just defining it once in our parent class Workout.

And it's going to do the following. So str method has to return a string. It doesn't print the string. This is a very important distinction. It returns a string that will eventually be printed when you call the print method. So the thing that I'm doing throughout this whole method, is to basically just build up my string to return. Return register is return string. And I'm building it just by concatenating it piece by piece with more and more stuff that I want to eventually print out.

So the output would look something like this. I'm basically printing out, on the console, sort of like the little square of a watch, very cute. So what am I composing here? The first bit, this thing in the red box, prints this line over here, just horizontal line that's some width long.

The next bit here, you notice, it grabs the icon data attribute, puts it here on a line along with a vertical bar and a bunch of spaces and a vertical bar. The next bit here, prints the kind of workout by accessing the kind data attribute. So either workout, or running, or swimming, whatever that string is. Prints it right underneath the emoji.

The next bit here, is composing the duration. So remember when we did the datetime object, just over here, when we were printing the duration where we just simply subtracted an end time minus a start time, It looked like this. I'm perfectly happy with that. That looks really nice. So let's just use that. So the get_duration just does the subtraction. It's a method inside my Workout class. And then we just keep composing that to our return string.

Next, we are going to figure out how many calories were burned in this Workout object. So again, we're grabbing the get_calories method, the return value from that method, however it may be calculated. So for this workout type, either we gave the value directly, or we let it estimate it from the duration of the workout. However it decides to calculate it, according to how this Workout object was made, that value gets put right there. And then the last bit is this last line down at the bottom.

So then we can create any kind of workout because all the child classes inherit all of the methods from the parent class. So of course all these child classes will inherit the str method of Workout. So no matter what kind of workout I'm creating, so here I'm just creating a regular workout, here a running workout, and here a swim workout. No matter how I'm creating it, they'll use the same str method.

So all of these will print it in this really nice format. The first bit will be specific to the kind of workout we have. The little emojis will be different because I've set those separately within the subclasses. The kind, as well, the label workout or running or swimming. The calories_burned and the duration will be calculated using the get_calories method and then the get_duration method. So again, in a nicely consistent way.

So I'll show you what this looks like in the actual code. Let me just comment that. So here I've got three workouts created. And then I'm just printing these three different kinds of workouts. And just to show you I'm not lying, see the swim workout doesn't have an str method defined and neither does the run workout. It just has a bunch of other stuff defined, but no str method.

So we're just using the str method of our parent, and then when we run it, it looks like this. So I've got a regular workout with their icon and label, running workout with their icon and label, and a swimming workout with their icon and label. Isn't that cute. All right, so we've made our own little digital thing.

So this begs the question, when can we use an instance of a class, of a subclass? Well, you can use an instance of this RunWorkout anywhere where you can use Workout because again, the way I think about it is, you say, well, a RunWorkout is a Workout, so anything I can do with a Workout, I should be able to do with a RunWorkout, or a SwimWorkout, or any of the subclasses.

But the opposite is not true. If I can do something with a RunWorkout, well, RunWorkout has a bunch of other specific things that it can do. Of course, a regular workout is not going to be able to do those specific things.

So let's think about these two helper functions. This one calculates the total calories given a list of workouts. And this one calculates the total elevation given a list of workouts. The code looks very similar for both. We're just iterating through the loop, grabbing each Workout object, and then we're calling the get_calories or the get_elevation on that Workout object.

So this will give me a number, and then I'm just keeping a running sum for the total elevation and the total calories and at the end, I return it. So again, the list here is important. These are Workout objects and Workout objects.

So what if I have a bunch of-- so here I've got two Workout objects and two running Workout objects. So these Workout objects are 30-minutes long. So using 200 calories per hour, these ones will each be 100 calories.

These running workouts are two hours long, so they will-- it doesn't actually matter. sorry, sorry. These running workouts are two hours long, so they're going to be 400 calories. Because these parameters here, correspond to the elevation.

And they correspond to the elevation because if we look at the way we define a running workout right here, this is the order of the parameters start time, end time, elevation value, calories. So when I pass in parameters, that need to go in that order. And I can't skip around. If I want one of them to be the default variable, then that has to be at the end.

So in this particular case, I've got these two running workouts at 400 calories because by default, I didn't actually pass in the number of calories. And then the elevation is 100 and 200. Yeah.

**AUDIENCE:** What if we wanted the default elevation and of the calories, would be put comma comma?

**ANA BELL:** Then if you wanted both to be default, then you just put nothing. You can't just leave an empty comma.

**AUDIENCE:** If you want the default elevation and [INAUDIBLE]

**ANA BELL:** Yeah, so then you would have to actually explicitly say, like, calories equals whatever you want. So at that point-- yeah, now that we're working with default variables, it becomes a little bit tricky. You can't go wrong with just saying, like, elev equals whatever you want it to be, calories equals whatever you want. And then you can do whatever you'd like in that case.

But in this particular case, we know our workouts are 400, and elevation is those values. So when we run total calories on all of the Workouts, no matter what kind of workout I have, it doesn't actually matter because Python will just grab calories for all of these workout types. So just sums that up.

Elevation, if I run it only on running workouts, Python will know what to do. Here's 100 and 200 because those running workouts have an elevation data attribute. But if I ask for the elevation for a running workout and just a regular workout, Python will spit out an error because as soon as it sees this workout one, it , says, well, what's the workout dot get_elev method. And it's going to say, I don't have a get_elev method for a regular workout. That's not something I know. Because that's something that we implemented in the child class.

So let's go through these together. And it's actually nothing to code just, to run. So it's just down here. So this is just kind of making sure you understand the order of operations. And I think one of the ones that was question here, where we actually passed in the number of calories, is at the end.

So when I create a regular workout-- oops, let me just remove that over here --what is the value when I ask Python to tell me the calories for this workout? At 200 calories an hour, what's the value here? Just yell it out.

**AUDIENCE:** [INAUDIBLE]

**ANA BELL:** 30 minutes? 100, right? What's the elevation when I ask Python to tell me the elevation for this object?

**AUDIENCE:** An error.

**ANA BELL:** Error, exactly, yep. Yep, because the Workout object has no attribute. It has no method get_elev. That's something specific to a running workout. OK, how about this one, RunWorkout here. So here I'm actually-- oops I didn't mean to do that. What happens if I grab the calories for w2?

**AUDIENCE:** 450.

**ANA BELL:** Yep, 450, yep it just grabs whatever's passed in, doesn't estimate. How about the elevation?

**AUDIENCE:** 70.

**ANA BELL:** Yep, there, again, perfect. Now let's create three kinds of running workouts. So here's one. What's the calories and elevation for this one? I'll just do them both together. So rw1. This parameter is the only one passed in. What does that correspond to, calories or elevation?

**AUDIENCE:** Elevation.

**ANA BELL:** Yeah elevation, remember our parameter list. Elevation comes before calories. So the elevation is 250, and the calories will be estimated based on whatever this is. So calories is first at 100. Elevation is 250. How about running workout 2? So here I've got 450 and 700 in that order. Which one is the elevation?

**AUDIENCE:** 450.

**ANA BELL:** Yep, 450, and calories is 700. So when we print it, I printed them backwards just to confuse us all. And then lastly, how about this RunWorkout 3? So here, to answer the question what if I wanted elevation to be default, but I wanted to pass in calories? So here I just say the name of my parameter there, and I give it an actual value. So clearly here, calories will be 300, and elevation defaults to zero. So just a little practice reading the specifications.

OK, so that finishes reusing the str method from the parent. Now let's override our superclass. So our improved Workout class, remember, has a get_calories method that estimates the calories based on the time that it took you to do this workout, whether it was a running workout or a regular workout.

But what I'm doing in this method, is I'm going to actually implement my own get_calories method inside the class definition for a RunWorkout. So here's my RunWorkout class definition. And I've got my own get_calories method. So when I run get_calories on a RunWorkout, Python will use this one. What is this one going to do?

So we're going to do something really cool. We're going to estimate the number of calories burned for a run workout based on a set of points, latitude longitude points. So what we can actually do, is we're going to pass in a list of tuples like this, which represents the route that we take, so in this particular case, I've got four places that I have jogged between. So these are my four latitude longitude points. So each tuple is latitude comma longitude.

So I can make this as precise as I'd like. But what I want this method to do, is to potentially, if the user does give me a set of latitude longitude points that they actually went through, to calculate the calories burned based on a class variable called calories_per_kilometer.

So given a set of these points, what I'd like to do is to calculate the total kilometers traveled between all of these latitude longitude points, multiply that distance, . That kilometer distance by the calories_per_kilometer. And use that as the estimate for the calories burned in this particular RunWorkout.

So this is how the code achieves that. So I've got another class variable that's only specific to this RunWorkout. So Workout does not know about this. Calories_per_kilometer is 100. And now I've got my own get_calories method here. It's overridden. So if we run this get_calories on a RunWorkout, it will use this one. And what does it do?

Well, if we don't give it any GPS points, if we don't give it a list of tuples there, Python will default to the else. What does the else do? Well, it says, hey, who's your parent? Run their get_calories method. So that's just estimating it based on the total time elapsed in this workout. That's our default parent.

But if the user got fancy and gave us a bunch of tuples, representing latitude and longitude points for all of their workouts, then we're going to do the following stuff. We're going to iterate through all of these pairs of GPS points, pair by pair. We're going to calculate the distance, given this latitude, longitude value. Add on to this running total for the total distance, and then return that total distance multiplied by this class variable, calories_per_kilometer.

So let me show you what this actually looks like because the only thing that is sort of still mystifying here, is this GPS distance. And this GPS distance is actually a function that's in this lecture helpers file, which is included in today's Python zip file.

And it's just from the internet. It's a way to calculate the distance traveled between two latitude longitude pairs. That's all it is. So it does some fancy stuff with sines and cosines and things like that to figure out the distance between these two lat long pairs. That's all it is.

So we're just running that function nicely down here to help us calculate that total distance. Beyond that, everything is pretty simple. It's just looking at consecutive pairs of these tuples, getting that distance plus this distance plus distances, and then multiplying by the cal_per_kilometer.

So in the end, what we get is something like this. So here let me show you. Here are two points, latitude longitude points. So I've got Boston and Newton, so here I've just got a straight shot. So I'm counting, getting very precise with blocks and things like that.

But if I create a running workout here with the start time, end time, elevation value, and now I pass in the route GPS points, this is another piece to my init method. I forgot to show you that. Sorry about that. So here's my init method for RunWorkout. I skipped that little bit. Last parameter here is to actually pass in some route GPS points.

And if I actually pass in those route GPS points, when I run the get_calories method, it tells me that I burned this many. And it calculates it based on that distance between Boston and Newton.

In the second example here, I don't actually pass in the value for the GPS points, so we're defaulting to just our regular calories function from Workout, which is to calculate it based on the start time and the end time. So from 1:35 to 3:57. That's why it's a weird not round number of calories. So I think that's also really cool, you guys, this function here.

OK, so these overridden methods, just to show you for completion's sake, how this Run Workout class looks, everything is the same as what we ended up with before. But now I'm going to reimplement my get_calories method, so now RunWorkout knows about a calories method. And I've also got this data attribute. Sorry, class variable, sorry, I always get messed up. This class variable cals_per_kilometer. And any RunWorkout instance will know about, of course, the cals_per_kilometer, as well as the cal_per_hour from our parent. Questions about that?

OK, we're building something really nice here. So I guess the question is, and I think you've probably figured this out, how do you know which method to call? Well, you just look at the object before the dot. You run a method your object dot method name, what's the thing before the dot? What is its type?

If the type-- like, for example, get_calories --if the type is running, you look to see if that class definition has a get_calories method. If it does, you use that. If it doesn't, and only if it doesn't, you look at your parent and say, does your parent have a get_calories method? If it does, you use that.

And if it doesn't, you look at the parents' parent. Does the parent's parent have a get_calories method? If it does, you use that. If it doesn't, you look at the parents parent's parent all the way up, you keep going all the way up the chain, until you get to the generic Python object. If the Python object type has a method named what you'd like to call, you use that, otherwise error, No such method was found anywhere within our chain of hierarchies up until the Python object.

All right, so that finishes overriding our get_calories method. And now we're going to do one more thing, which is to add something new to RunWorkout that didn't actually exist in Workout Although, I guess I am implementing it in Workout. So it's not actually adding new. But we're going to override it anyway.

So the class Workout, let's say that we want to compare two workouts together. So to do that, we're going to implement the Dunder method double underscore eq double underscore. And this will allow us to compare two running Workout objects, or two Workout objects, or running and Workout objects using the double equal sign. So w1 == run w2 or whatever.

So we can use the double equal sign to compare objects of our type. So again, my decision for comparing these two objects, Workout objects is to say, well, first, let's compare the types. So if I'm comparing a workout versus a running workout, right off the bat, they're not going to be equal.

So first of all, the type of this object should be the same. So I should be comparing workouts with workouts, running workouts with running workouts, or swims with swims. And I also want every one of the other data attributes to be the same. So the start time, end times, the kind, and the get calories. So as long as all of these things are the same I'm going to say that these workouts are the same or equivalent.

So this is the equal method in my workout, and then, in my class Workout, I can actually override that method. So this should actually be add override, just like the other one. And then RunWorkout, I'm going to override the equal method, but I'm going to do it in a very modular Pythonic way.

I'm going to say that a RunWorkout is going to be the same as another Workout if everything in my parent is the same. So here, I'm just calling the super() method saying Workout dot double underscore equal double underscore other. So with this little bit here, this line here, just the super() dot double underscore equal double underscore other, this compares all of these things. So I don't need to rewrite that in eq the method of RunWorkout.

And I can clearly see what else in addition to regular Workout comparison, I need to have happen for them to be equal. I also want the elevations to be equal. That's the other data attribute.

So you can see now how nicely modular this code looks. It's very clear what differentiates a RunWorkout to a regular Workout with this slide. Questions about this? OK.

**AUDIENCE:** Should [INAUDIBLE] to continue to end.

**ANA BELL:** Yeah, exactly, yeah, so this should all be on one line. But the backslash actually just breaks up the line into multiple lines for visibility. So in the code, here's a bunch of Workouts. And we can run some of them. But you can see why they're true or false.

So here, w1 and w2 are not the same because the calories_burned are different. They're both regular workouts. They both have the same start and end times, but the calories_burned are different. So this prints false, just equality on these workouts.

And then here's a true one. w2 is equivalent to w3 because the start and end times are the same, the length is the same, and the calories_burned are the same, w1 and w3. Or sorry, w2 and w3, these two, all right. This one just used the default value, but that default calculated values was calculated to be 100 because it's a 30-minute workout, anyway. So you can go through some of the other ones on your own.

I guess the other interesting one is this w1 with rw1. Everything about this is the same, calories_burned. Everything is the same, except for the fact that they are different kinds of workouts. One is a run. One is a regular. So we run that as false. Other questions, or any questions?

OK, so last slide, this is the last lecture on object oriented programming. Hopefully, it gave you an idea for how to create your own objects. And this last example, specifically, showed how we can just improve it a little bit at a time to make it be this really cool thing.

We added a way to estimate calories. We added a way to estimate calories using GPS points, and we just did it incrementally. So you don't want to do that right off the bat. Just write a little bit at a time, and in the end, you can write a really cool object type.

Now that you know how to create your own object types, you can create objects using other objects. So some of the data attributes for something more complex, could be a Workout object, something like that. But it's possible to overdo it. Especially now that we're not writing super complex classes, it's possible to overengineer.

And when you overengineer, it becomes kind of annoying to just keep scrolling back and forth to this init, to that init to figure out what methods were in this class, what methods were in the other class? And so if you can achieve it using just one object type, or maybe just a function, no need to create your own all these complicated object types that build upon object types.

But I just wanted to show you that it is possible, especially as you might be building more complex things in future classes, things like that. It is possible to write really complex classes that don't look so bad because you're building upon code that you've already written. So now we've got these ideas of abstraction, modularity, information hiding that all work together to help you achieve this really cool object or cool class or cool program.

OK, so the next set of lectures, we're going to leave programming for a little bit. And we're going to look at figuring out how to write efficient programs and how to figure out whether our programs are efficient or not and things like that. So we're going to go into a more theoretical side of computer science.