Python object-oriented programming

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Two of the best things about the Python language are its power and its simplicity. You can accomplish quite a lot in Python, while keeping your programs relatively small, and easy to digest. But as your projects get larger and more complex, you'll soon need a way to keep your programs and data organized, extensible, and easy to adapt and change, as new team members are brought on to the project. Hi, I'm Joe Marini, and I've been building software for some of the best-known companies in Silicon Valley for more than 30 years. In this course, we're going to learn about the object-oriented programming features of Python, and how they could be put to work in your projects. We'll start with the basics of defining and using classes and objects, then we'll learn about some of the more advanced features, like abstract base classes, and how to implement interfaces. We'll also see how to use some of Python's unique features, like the "magic" class methods you can use to tightly integrate your objects with the Python language itself, and we'll see how to use Python's data classes to build objects that efficiently store and manipulate information. These techniques will enable you to build programs that are modular, resilient, and extensible. Let's get started learning about object-oriented programming in Python.

### Object-oriented programming refresher

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Before diving into the details of how Python implements object-oriented programming techniques, let's take a quick refresher of some object-oriented programming principles. So let's begin with some of the reasons and motivations for using OOP in the first place. Now, Python does not require you to use objects or classes when creating your programs, even though everything within Python itself is an object. If you want to create individual functions and variables and data structures and so on, then you're free to do so, and Python will happily execute your code as long as it's syntactically correct. As programs become more complex, however, it becomes increasingly difficult to keep everything organized and to ensure that one part of your program isn't having unintended side effects on another part. Using object-oriented programming techniques helps organize and structure programs. This makes them easier to plan, maintain, and learn by new programmers, even if that's you several years in the future coming back to your code after awhile. With this approach, you build a program that groups together data and the functions that operate on that data into one place. This helps reduce bugs and helps ensure that only the parts of the program that need access to that data have it. Object orientation also promotes the building of modular programs, meaning that individual parts of the program don't need to know how other parts work internally. So down the road if part of the program needs to be updated or replaced, it can be done with minimal disturbance to the code in the rest of the program. It also helps isolate different parts of the program from each other. So if a problem occurs in one part of the program, then other parts of the program can continue to operate. All right, so now let's take a quick review of some of the terms we'll see in the rest of the course. You can think of a class as a blueprint or a template that describes how to create a specific kind of object. Methods are a fancy way of referring to functions that are part of a class, while attributes are a way of referring to variables or properties that are part of a class definition. An object is a specific instance of a class or a type. So for example, the number one is an instance of an integer type. The term inheritance describes how you can arrange classes in a hierarchical fashion where a particular class can inherit the capabilities and data of another class, and we'll see how this works in Python later in the course. Another important term to understand is composition, which refers to building complex objects out of other objects. And again, we'll see an example of this later. All right, so now that we had a quick refresher on object-oriented programming terms and principles, we can get started with the rest of the chapter and see how Python puts these into practice. If you're still unsure about some of these terms and principles, I would encourage you to review the what you should know video at the start of this course for some direction on where you can learn the basics before continuing.

### Basic class definition

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Let's begin by seeing how to create a basic class definition in Python along with how to instantiate the class and then access parts of it. So here in Visual Studio Code I'm going to go ahead and open up a folder and I'm going to open my exercise files folder. All right, and then in chapter one I'm going to go ahead and open up the definition\_start file. So let's go ahead and get started. The class keyword is used to define a new class. So let's go ahead and start there and I'll go ahead and make a class called book. Now you can optionally add parentheses after your class name but they're only really required if you're going to indicate that the class inherits from another class. And we'll get to that later in the course. So since our class just exists on its own, we can leave those parentheses off. So now to complete the class, I can just add the pass statement in Python and this is perfectly legal. I mean this is a syntactically correct class definition. I mean obviously it doesn't do anything but I can now create instances of this class simply by calling in the class name as a constructor function. So I can write b1 is equal to book. And that will create a book object. In reality, however, our classes will contain methods and data attributes. So let's fill out this class a little bit more. So first I'm going to delete pass and I'm going to override the built in init function. So I'm going to write a function like I normally would only I'm going to do it as part of a class and I'm going to override the init function like this and it takes two arguments. We'll I'm going to give it two arguments called self and title. And then inside this function I'm going to write self.title is equal to title. All right. So the init function is one of Python's special functions for working with classes. When the class is created like we see here on line 11, the init function is called to initialize the new object with information. And it is called before any other functions that you've defined on the class. Now some people mistakenly call this the constructor function because other languages like Java and C Sharp have a similar function that performs a similar purpose. But it's more accurate to say in Python that this is the initializer function and not really the constructor. The object has already been created, or constructed, when this function is called. So you know it's safe to start initializing attributes. So you can see here that I've passed in an attributed named title. And this is going to be one of the arguments to the constructor function. And then there's this other function here named self. And we'll get to that in a little bit. All right, so let's go ahead and update our code to call the book object correctly. So now I need to pass in some titles. So I'll pass in brave new world and I'll create another one called war and peace. All right, so you may have noticed that the init function takes two arguments. But I'm only supplying one when I create each book. And that's because whenever we call a method on a Python object the object itself gets automatically passed in as the first argument. And the word self by the way is just a naming convention. You don't need to call it that, but most Python programs follow this convention. So then inside the init function you can see that I'm creating a new attribute on the object called title and it's being used to hold the title of the book. So once I've done this I can then access the value of this property using dot notation just like any other object. So first I'll print out b1 and then I'll print b1.title. All right, so let's go ahead and save and let's run this. Now here in Visual Studio Code because I have the Python extension installed I can run this right here in VS Code. Let's go ahead and do that. And you can see that I'm printing out b1 and that's just the name of the book object itself. You can see it's the class name and the memory location where it is. And then you can see the next line prints out the title. Right here on line 18. And that's this right here. So if you're not using Visual Studio Code, you'll need to run this directly from the command prompt. So let's go ahead and bring up the terminal. And I'm going to go ahead and go into my exercise files folder which is on my desktop. And we'll go into chapter one. And then I can just run it directly. I can write Python and this is called definition\_start. And you can see the same results there. Okay, so that gets us up and running with creating Python classes. In the rest of the chapter we'll see how to work with other kinds of methods and properties and how to identify classes and objects.

### Instance methods and attributes

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In the previous video, we saw how to define a basic Python class and create an instance of it. Now let's see how to create instance methods and attributes for our class. So to do that let's go ahead and open up instance\_start.py and you can see that it's pretty much where we left off from the previous video. So here's my book class and the init function and we can create a couple of book objects. So let's start by filling out the init method a little bit more since books tend to have more information about them than just their title. So let's go ahead and add arguments for author and page count and price. And then I can go ahead and add these. So I'll have self.author and we'll add pages and then we'll add price. All right. Now each one of these attributes is called an instance attribute because the value that it holds is unique to each instance of the object that it's declared on. And the same goes for instance methods. So so far we only have one which is this init function right here but we can add more. So let's add a method to return the price of the book. And again, remember that every instance method takes an object as the first parameter. So I'll name my function getprice. And the objects are the only parameter in this case. And I'll go ahead and return self.price. All right, so now we can update the code where we create these books. So let's add the authors. And War and Peace was written by Leo Tolstoy. And it's a big book, it's got 1225 pages. And we'll set that to be 39.95. Cather in the Rye was written by JD Salinger. And that's a little bit smaller. It's got 234 pages. And we'll set that to be 29.95. All right. So now let's try making the call to the getprice method. So I'll print b1.getprice. All right. Let's go ahead and run the code we have and I'll just run it right here in Visual Studio Code. And you can see right there is the price, right, 39.95. Okay, so let's go back to the code. Now we're not limited to creating instance attributes just within the init function. We can do it elsewhere in the object as well. So let's add another function that specifies a discount for a book that's on sale. And I'll call that function setdiscount. And that will take the object and the discount amount and we'll set self\_discount equal to the amount. Now notice how I have an underscore in front of the attribute name. The reason for this is to give other developers who use this class a hint that this attribute is considered internal to the class and should not be accessed from outside the class's logic. Other languages like Java or C# have a way of declaring attributes that are only intended to be used within the logic of the class where they're defined. But Python doesn't have a formal way of enforcing this. The underscore in the name basically tells other programmers hey, this attribute can't be relied on to stay this name so don't use it in your code. So now that we've defined the attribute, we can use it elsewhere in the class. Let's modify the getprice function to use the discount if it's defined. Now since discount was not defined during the init function we can't rely on it being present because setdiscount might not have been called yet. So we need to test for it by using the has attribute function. So I'll call if has attribute and I'm going to test my object to see if the discount attribute is present. So if it's there I'm going to return self.price minus self.price times self.\_discount. Otherwise just return self.price. All right. So let's try that function out. So we'll go ahead and print book two's price. And then we'll set the discount to be 0.25 percent. And then we'll try printing the price again. All right, let's go ahead and run that. Okay, so here's the price of book one. Here's the price of book two before the discount. And then here's the price of book two after the discount. All right. So one last thing I want to demonstrate, and let's go back to the code. Now as I mentioned earlier, the underscore is used to indicate that an attribute or a method for that matter is intended to only be used by the class. Now if you use a double underscore as the start of an attribute or method name, then the python interpreter will change the name of that attribute or method so that other classes will get an error if they try to access it. So let's look at that. Here in the init function I'm going to add an attribute named secret and I'm going to name it underscore underscore secret. And I'll give it a value that just says this is a secret attribute. All right, so now let's try to access that attribute. I'm going to print b2 dot underscore underscore secret. All right. So let's go ahead and run this code. Let's go ahead and comment these out first. So let's run that. And you can see that I'm getting an error because that property can't be seen outside the class. You can see it say attribute error and then the book object has no attribute double underscore secret. However, this is not a perfect mechanism. So the way that Python does this is by prefixing the name of the attribute with the class name. This is called name mangling. The reason for this feature is to prevent sub classes, which we'll learn about later, from inadvertently overriding the attribute, but other classes can subvert this simply by using the class name. So if I go back to the code and just simply put underscore book in front of this and then save it, I'm going to run this again. All right, well there's the print out, right. It says now this is a secret attribute. You can see now that I can access the property. So it's not a perfect solution, but you can use this approach to make sure that subclasses don't use the same name for an attribute that you've already used. Now in some cases that's exactly what you want. You do want subclasses to overwrite things sometimes. But if you need to make sure that they don't, then the double underscore can prevent that.

### Checking instance types

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Sometimes it's useful to be able to check, during runtime, what type or class a given object is, and in Python we can do this with the type function and the isinstance function. Let's open up typecheck\_start to see how this works. So here in my code you can see that we have two classes defined. One is called Book and one is called Newspaper, and there's also some code that creates some object instances of each type. So, let's first use the type function to see what type a given object is. So I'll print and then call type and then pass in b1 and I'll copy that and try the same thing with n1, which is a Newspaper. So b1 is a Book, n1 is a Newspaper, and I'm going to go ahead and run this. And you can see that when I run this that b1 is of type Book, right here Class Book, and the other one is called Class Newspaper. Now the type function is useful because you can use it to compare two different objects to see if they are the same type. So, let's go back to the code and I can do something like this. I can say print and call type of Book One and check to see if it's the same type as b2, and just for completeness, let's try comparing Book One with Newspaper Two, all right. So, let's go ahead and comment this and now let's run it. And you can see that the two Book objects are of the same type, whereas the Book and the Newspaper are obviously different types. So, to see if a given object is an instance of a particular class, it's much cleaner to use the isinstance function rather than trying to parse the type string, which you can do, but there's a function that's already built in for this. So let's go ahead and try that out. I'm going to print out and I'm going to call isinstance and I'll pass in the Book object and then the Book class. Not in a string, not in quotes or anything, just the actual class itself, and then I'll do the same thing. I'll check to see if n1 is a Newspaper and then I'll try that one more time, and I'll try checking n2 against the Book class. Now, it's also possible to use this on subclasses and we haven't really covered inheritance in subclasses yet, but in Python everything is a subclass of the Object class, so we can try that as well. So I'll print isinstance and check the Newspaper against the built in object type. Before I run this let's go ahead and comment these out, so we have some clean output. So I'll save and I'll run and you can see that in the first case, let me scroll the code down here. So, b1 is an instance of Book, n1 is an instance of Newspaper, n2 is not an instance of Book, but n2 is an instance of Object of which every Python object is a type because all classes in Python derive from Object.

### Class methods and members

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Let's finish up this chapter by examining two more kinds of methods and attributes. So we're going to look at class and static methods and attributes in this example. So here in my editor I have opened up the class\_start.py file. And there's already some code filled out for the book class. So I'll just collapse this down. So we've already seen how the init function works. And we've already learned about instance functions. And we know for example that setTitle is an instance function that sets the book's title in this case. And the title attribute itself is an instance attribute because its value is associated with each instance of the object that gets created from this class. So now let's start with class-level methods and attributes. These are different from their instance versions because they are shared at the class level across all instances of that class. Now that might sound a little confusing so let's take a look at a concrete example. Suppose we wanted to make sure that each book object was assigned a specific book type when it was created such as hard cover or paperback or ebook. Now we could define an instance attribute that enumerates these values but since it's going to apply to all the book objects it would make more sense to just define it once. What I can do is create an attribute at the class level outside of any of the instance methods and I'm going to use all caps here to indicate that this is a class attribute. And I'll call it Book\_Types and I'm going to initialize it to a tuple that contains hardcover, paperback, and ebook. So then I can have the init method take the book type parameter and then I can check to make sure that the given type is one of the values that the class allows. So I'll write if not and then booktype in and then to refer to the class variable I use the class name. So it's book.book\_types. Then I'm going to raise a ValueError and we're going to use my format string here to say that booktype is not a valid book type. Otherwise we'll just go ahead and set self.booktype equal to booktype. Okay. Let's also add a class method that returns the booktypes list. So to do this I create a method and I use the classmethod decorator. And I'll define the method called getbooktypes. And this works on a class instance not an object instance. So I'm going to return the classes BOOK\_TYPES constant. And then let's go ahead and print out the result of our class attributes. So we're going to go ahead and call print and I'll write book types and once again I use the class name to call the class method. All right. So actually I need to change my object code here where I create the books. So let's make the first one a hard cover. And then let's make the other one a comic. Okay, so let's go ahead and run. And here in my terminal you'll see that we successfully print out the book types. So that's this line of code right here. But then I get an error, right, because it says COMIC is not a valid book type. It's not one of the pre-defined book types that I've defined. So let's go ahead and change that. Let's make this paperback. All right. And then let's run again. And now we see that everything is fine. Okay, so let's go ahead and take a look at static methods now. So where instance methods work on specific objects and class methods work on the entire class, static methods are different in that they don't modify the state of either the class or a specific object instance. And I should point out here that there aren't really that many great use cases for static methods. They are useful, however, for scenarios where you don't need to access any properties of a particular object or the class itself but where it makes sense for the method to belong to the class. In other words it's a good way of name spacing certain kinds of methods. Whereas you might in another way create a global function, this is a way of taking a global function and just putting it in the classes namespace. So one good example of static methods is to implement a singleton design pattern. And remember, the singleton design pattern let's us make sure that only one instance of a particular variable or object is ever created. So let's imagine that we wanted to have book class to be responsible for keeping track of a list of books. Now we could create a global variable to hold all the books but it might be a better approach to encapsulate that behavior within the book class. So what I'm going to do here is create a double underscore attribute named booklist. And remember, the double underscore essentially makes this a private variable. And we saw this previously in the chapter. And then I'll define a static method that exposes this property to the consumers of the book class. So I'll initialize this to none and then I'll create a static method to access it. So to create a static method I need to use the static method decorator. And then I'll define my method. I'll call it getbooklist. So then I can check to see if the booklist attribute is null and create a new list. So if book.\_\_booklist is none then I'll just create a new one. And if it's already been created, then we'll just return the existing one. So now let's use this feature to add our books to a list. So I'm going to scroll down to the bottom here and I'll write some code. So I'll create a books variable and I'll assign it to the result of getbooklist. Now remember, this is a static function. It's on the book class, but it doesn't actually modify the class's state. So then I'll write thebooks and I'll append book one and then I'll append book two and then we'll print out thebooks. All right, so let's go ahead and run this. And here in the output we can see that our books are being added to the list. And remember, we've ensured in our logic that only one of these will ever be created. Right, so if it's none, then we create it. Otherwise we just return the existing one. So we've namespaced what would otherwise be a global function to our booklist and this is one of the good uses of a static method. There are some other ones, but more the most part they're really useful for namespacing what would otherwise be a global function into the namespace of a class.

### Understanding inheritance

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] One of the core concepts of object-oriented programming is the notion of inheritance, and in this example, we're going to see how that works in Python. Inheritance defines a way for a given class to inherit attributes and methods from one or more base classes. This makes it easy to centralize common functionality and data in one place instead of having it spread out and duplicated across multiple classes. Let's go ahead and open up our inheritance\_start file, and in this example, you can see that I have three classes. There's a book, magazine, and newspaper class. Each one of these classes represents a type of publication, and each of them has a set of attributes that are relevant to that publication type. So books have a title and a price along with the author's name and their number of pages. A newspaper also has a title and a price, but they are published on a periodical basis and have a publisher instead of an author. Magazines have a title and price too and have a recurring publishing period and a publisher, and you can also see down here that there's some code that creates each kind of object, and then, accesses some data on each of them. So let's go ahead and run this as is. Alright, and you can see that in the output, we have the book's author, which is this line of code right here, we have the newspaper publisher's name, and then the prices of each one. Now at the moment, each of these is a standalone implementation of it's own class, but let's go back to the code, and you can see that there's a considerable amount of duplication among the data that each class holds. So, for example, all three classes have attributes for title and price, and the newspaper and magazine classes also have the same attributes for period and publisher. So we can improve the organization of these classes and make it easier to introduce new classes by implementing some inheritance and class hierarchy. So let's start with the most obvious duplication, which is the title and price attributes. So one way we can handle this is by defining a new base class called publication, and then, have that class define some common attributes. So I'll put my init function in here, and we'll give that a title and price, parameters, and then we'll just set self.title equals to title, and then the same thing for price. Alright, so now, we can fix the book class and have it inherit from the publication class, and then we're going to put the name of the base class in the parenthesis here, and now what we need to do is call these super classes init function, and then, we can just take off the title and price, and then just have the book specific attributes in the book class. Now we could do the same thing with newspaper and magazine classes, but there's some duplication here too. Both of these classes have period and publisher attributes, so that's a pretty good hint that we can collect those in a superclass too. So let's go ahead and make another base class and we'll call this one periodical, and we'll have that class based on publication, and then, once again, we'll create the init, and a periodical will have it's self, title, price, period, and publisher, so then we call the superclasses init for the title and the price, and then, we'll have the periodical class to find the period and the publisher. Alright, so let's go ahead and save that. Okay, so now, we have a class hierarchy with publication at the top, and book inherits from that, and then we have periodical, and then, that inherits from publication as well, and now we have to fix magazine and publisher to inherit from periodical. So let's go ahead and do that, so now we have the base classes, and what we're going to do is call the superclass to init each of these, and we'll pass in the title, price, period, and publisher, and then we no longer need these, and then I can just do the same thing here for the newspaper class. Alright, so now, we should be able to run our original code that creates these objects and accesses the data without any changes, so let's go ahead and try that, and when I execute the code, you can see that the output is the same as before. So we're getting the same results, but with much better code organization, which is one of the main benefits of inheritance. So I can now add properties that are specific to each kind of publication just in one place and I only have one place to edit them if I want to change the names of any of these attributes going forward.

### Abstract base classes

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] So now that we've seen how inheritance works in Python, let's move on to a related topic called abstract base classes. There's a fairly common design pattern programming where you want to provide a base class that defines a template for other classes to inherit from, but with a couple of twists. So first, you don't want consumers of your base class to be able to create instances of the base class itself. Because it's just intended to be a blueprint. It's just an idea. And you want subclasses to provide concrete implementations of that idea. And then second, you want to enforce the constraint that there are certain methods in the base class that subclasses have to implement. And this is where abstract base classes become really useful. So let's go ahead and open up the abstract underscore start file. Let's imagine that we're building a drawing program that lets the user create different kinds of two dimensional shapes. And we want the program to be extensible so that new shape types can be added. So you can see here that I've defined a base class called graphic shape, and it has a function called calcArea that is currently empty, right? There's no implementation here. And then I have two subclasses, circle and square, both of which inherit from graphics shape. So the scenario here is that we want each shape to inherit from graphic shape. We want to enforce that every shape implements the calcArea function, and we want to prevent the graphic shape class itself from being instantiated on its own. Now, if I run the existing code that I have here, you'll see that none of these constraints are currently enforced. So I can instantiate the graphic shape. And if I run this, you'll see that the calcArea function returns nothing, right? Because we didn't override that in the subclasses. So to fix this, let's go back to the code, I'm going to use the ABC module from the standard library. So what I'm going to do is from ABC, I'm going to import ABC. And I'm going to use abstract method. Alright, so the first thing I'm going to do is have graphic shape inherit from the ABC base class. And that stands for abstract base class. Then I'm going to use the abstract method decorator to indicate that the calcArea function is an abstract method. So this tells Python that there's no implementation in the base class. And each subclass has to override this method. So now you'll see if I run this, well, now I get an error from trying to instantiate the graphic shape right? It says can't instantiate abstract class graphic shape. So let's go ahead and comment that out. All right, and now let's try it again. Now I'm getting another error, right? It says that my subclass didn't override the calcArea method. So now I need to fix that too. Let's go back to the code. And so for the circle, I'm going to write def calcArea. And that's going to return 3.14, which is pi, right? Times the radius of the circle squared. And then for the square, I'll do the same thing. And that's going to return self dot side times self dot side, okay? All right, so now I've satisfied all the conditions. I'm no longer trying to instantiate the graphic shape by itself, and now both of my subclasses, override calcArea. So let's run this again. And now you can see that everything is working. So abstract base classes can be a very useful tool for enforcing a set of constraints among the consumers of your classes. So it's worth taking the time to experiment with these and understand their benefits.

### Using multiple inheritance

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Unlike some other popular programming languages, Python lets you define classes that can inherit from more than one base class. This is called multiple inheritance and while it can be a useful tool, it can also cause a lot of problems if you don't use it carefully. So let's open up the multiple\_start file and see how it works. So here in my code I have two classes, Class A and Class B, each of which define an attribute. So, in Class A we have foo and in Class B we have bar. Then I have a third class named Class C, which lists both Class A and Class B as base classes, separated by a comma, and this is how you inherit from more than one class at the same time. So, let's go ahead and add a method to Class C to print out those attributes. So I'm going to call the method showprops and I'm going to print out self.foo and then I'm going to print out self.bar, each of which is inherited. And then let's call that method after the C object is created. All right, so let's go ahead and run this code. And in the output you can see that the two properties are being printed, right, so there's foo and there's bar. So everything seems fine. But what happens when each of the two superclasses define the same attribute? So, let's go ahead and add an attribute to each one. Let's have self.name equals Class A and then let's have self.name in B equal to Class B, and now let's print this out in the showprops method. So now, I'll print out self.name. So now, let's run the code again. So we'll save and we'll run, and now we see foo bar and okay, I guess Class A is somehow the winner? But why is that? So, when you call a method or access an attribute in Python the Python interpreter uses something called the method resolution order to look it up in the class. So the lookup starts in the current class, in this case Class C, which doesn't define the name attribute. So then Python looks in the superclasses in the order in which they are defined from left to right. So since Class A is listed first, that's why we're seeing the Class A string in the output. So if I change the order to be B and then A, and then I'll save and then I'll run the code again. And so now you can see that the function prints out Class B because that's the one that's being looked up first because it comes first in the order. Now, it turns out that you can actually inspect the method resolution order by looking at a special class attribute called mro. So let's go ahead and add that. I'm going to print the C classes underscore underscore mro attribute and now when I run this you can see that the method resolution order is first it's Class C, right, then it's Class B because that comes first in the order, then Class A because that's next, and then the Object class which is the implicit superclass for all objects in Python. So this added complexity is one of the main reasons why you don't see a whole lot of multiple inheritance in real world projects, but there is a place where they actually are very useful and that is in implementing a programming construct called an interface, and we'll see how that works in the next video.

### Interfaces

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this example, we're going to see how to use a combination of multiple inheritance and abstract base classes to implement a type of programming feature called an interface. Some languages like C sharp and Java, provide this feature as a built-in part of the language. But Python doesn't have explicit language support for this. You can think of an interface as a kind of promise. By implementing an interface, a particular class makes a promise or a contract, as it's often called in software engineering, to provide a certain kind of behavior or capability. So, let's go ahead and open up the interface\_start file, and we'll see how this works. So, you can see that this is now a simplified version of the abstract base class example that we used earlier with the shapes. So, suppose we wanted our concrete shape objects to be able to represent themselves as JSON. Now, we could just make that a part of the graphic shape base class, but if we had a variety of other objects in our program that we also wanted to have that function, then we would need to add that to each of those base classes as well. And that's just a lot of needless duplication. So, what we can do instead is create another abstract base class, and we'll call it JSONify. So, let's go ahead and create that. And I'll have it inherit from the abstract base class. And then we will have this class to find a single abstract method called toJSON. So, we'll define toJSON, and put an empty implementation in that class. So, now notice, this method does not provide any implementation itself. It just defines the name of the method. So, we can then add this new class to the definition of the circle class. Not the base class, not the graphic shape, we're going to put it on the circle. So remember, this has the effect of requiring that the circle class has to override and implement the toJSON abstract method. Otherwise, it's an error. So, let's go ahead and run this, and you'll see, sure enough, we get the error because we haven't implemented the toJSON method. Let's go back and add that to the class. And I'll put that below the calc area. I'll name it toJSON. And I'm just going to return a formatted string, and we'll put in some JSON here, and then we'll close that off. Alright, and then I will call that function down here in the exercise code. So I'll write print c dot toJSON. Okay. So, what I've essentially done is created a method that I have to override, called toJSON, and I've mixed that in on my subclass by having two base classes here, then I override that method to provide a string that's just a JSON representation of my object. So, it's going to have, actually it shouldn't be square, it should be circle. Whoops. There we go. Okay, so go ahead and run this. Alright, and there you can see in the output, we have the results of the calc area function, and then we have the results of the JSON right here. So, what we've essentially done is created a very small, focused class, right here called JSONify. that we can now use whenever we want another class to be able to indicate that it knows how to represent itself in JSON. We didn't have to modify the graphic shape base class in order to do this, which gives us the flexibility to apply this new class, which is serving the function of an interface, anywhere it's needed. So, interfaces are really useful for declaring that a class has a capability that it knows how to provide. And even though Python doesn't have explicit language support for it, it's flexible enough to be able to implement it with abstract base classes and multiple inheritance.

### Understanding composition

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Earlier in the chapter, we learned about how Python implements the concept of inheritance, to create class hierarchies. In this example, we'll see how we can use a concept called composition, to create complex objects out of simpler ones. So you might recall that inheritance models and is type of relationship. So here in this class diagram, a book object is a publication because it inherits from the publication base class, and picks up all the attributes and methods from the base class as well. So we can also say that magazine is a periodical, and we can see that magazine is a publication because again, it has the same common base class. Composition works a little differently. When using composition, we build objects out of other objects, and this model is more of a has relationship. So in the diagram on the right, the book object, has an author object, which contains information about the author. Rather than defining all of the author related information, directly within the book class hierarchy. This type of model lets us extract distinct ideas, and put them into their own classes. Now, inheritance and composition are not exclusive, you can combine both depending on what your applications needs are. So let's apply this concept in some real code, to see how it works. Well here in VS code, let's go ahead and open up the composition underscore start. And I've defined a book class that's a little bit different than the one we've been using so far. So there's the title and the price, along with some author information, the first and last name, and there's an attribute to hold the list of chapter information. There's also a method to add chapters to the book, and it takes the name of the chapter and the pages count and just adds a tuple to this collection right here. Now, this particular class definition is all fine and good, but it's pretty monolithic. There are pieces of information like the author, and maybe the chapter information that might make sense to treat as separate entities. It's not hard to imagine a scenario where we might want to work with just a group of authors or get information about specific book chapters. So we can use composition to separate these discrete pieces of information from the overall book object. So let's start by extracting the author information into its own class. So I'll make a class called author, and we'll go ahead and implement the init method, and that will take a first and last name, and we'll just go ahead and store that on some attributes. Okay? And while we're at it, let's go ahead and give that author class A nice string representation, and we'll learn more about this when we get to the chapter on magic methods. But I'm going to override this string method, and I'm going to return just a nicely formatted string consisting of the first name, and the last name. So then I have to modify the book class, to take an author object as an argument. So I'll go ahead and replace this with author, and that's going to default to none. And then I can get rid of these two, and replace that with self dot author equals author. So now, we've created a relationship where a book has an author associated with it, instead of keeping that implementation details of the author data, wrapped up within the book class. We can do the same thing with the chapter information. So let's go ahead and create a separate class for the chapters. And I'll create chapter objects and I'll, implement the init method for that. And that's going to take a chapter name and a page count. And we'll just go ahead and set those properties. All right. And now we have to modify the add chapter, method in the book class 'cause it no longer takes separate name and pages, it takes a chapter, and our chapters collection, just adds that chapter to the list. And so here again, we've now created a relationship where a book has a collection of chapter objects. We can even add a new method, and we'll call this get book page count, whereby the book can calculate its own page count. So we'll start off with zero and we'll just iterate over each one of the chapters, and then we'll add up the chapter page count, and then we'll return that result. All right. So now let's clean up the code that uses these classes down here. So for the book, I now need to pass an author constructor in here, so I'll just go ahead and create an author. All right, and we'll pass that in as the second argument here. Alright, and now we have to add chapter objects to the book. So this is going to become a chapter object. Alright and I'm going to just copy and paste that each time, and we'll close that off. Looks like they've added an extra one. There we go. Okay, so now we get some nice separation of responsibilities. So for example, printing the full name of the author, is done within the author class. So we can print out the books author, and we'll leave the title alone. And calculating the book size is done by using the data, that's in the chapter class. So we'll print out the page count. Alright, so it looks like everything is now in place. We've got our extracted chapter and author objects, and we've updated the book class to use that data instead. So let's go ahead and run our updated code. And you can see, right here, so the title is unchanged, and now here is the author information being printed out from the author class, and there is the total page count. So what we've done is taken a monolithic class definition, and made it more extensible and flexible, by composing it from simpler class objects, each of which is responsible, for its own features and data.

### What are magic methods?

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this chapter, we're going to spend some time learning about Python's so-called magic methods, which are a set of methods that Python automatically associates with every class definition. Your classes can override these methods to customize a variety of behavior and make them act just like Python's built-in classes. Now there's quite a few of these methods, and I'm not going to cover all of them. Instead, I'm going to demonstrate the ones that are most useful and commonly employed. Using these methods, you can customize how your objects are represented as strings, both for display to the user and for debugging purposes. You can control how attributes are accessed on an object both for when they are set, and for when they are retrieved. You can add capabilities to your classes that enable them to be used in expressions such as testing for equality, or other comparison operations like greater than or less than. And then finally we'll see how to make an object callable just like a function, and how that can be used to make code more concise and readable. Features like these are what gives Python its flexibility and power, and in this chapter, we'll see examples of how they can be put to good use.

### String representation

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] The first set of magic methods that we're going to learn about are the ones that Python uses to generate string representations of objects, and we saw a little bit of a peek of this in the prior chapter when we worked on object composition, but we're going to see much more of it now. So let's go ahead and open up the magicstr\_start file, and you can see that I have my book class defined with some properties, along with a couple of statements to create some book objects. So there are two magic string functions, one is called str and one is called repr. The str function is used to provide a user-friendly string description of the object, and is usually intended to be displayed to the user. The repr function is used to generate a more developer-facing string that ideally, can be used to recreate the object in its current state. It's commonly used for debugging purposes, so it gets used to display a lot of detailed information. So these functions get invoked on an object in a variety of ways. So for example, when you call the print function and pass in the object, or when you use the str or repr casting functions, these methods will get called. So let's run our code as it currently is before we override these functions. You can see that I'm creating two book objects and then printing them out. So let's go ahead and run this. And so here, in the output, you can see that when I print each object, I just get a vague string that identifies the class name and its location in memory. So let's make that a little bit better. Let's go ahead and add the str function, and you can see that these are double underscore function names, indicating that they are Python magic functions. So when I override the str function, I get to decide what the string representation looks like. So I'm going to return a formatted string, in this case it's going to be self.title by, and then it's going to be self.author, and it costs self.price. All right, so now let's rerun the code. Now you can see that when I print these objects out, there's a much nicer string representation of each book object, containing their realtime data. All right, so let's go back to the code. Now let's add the repr function, and it's going to return a slightly different string. So this is going to return a formatted string as well, and just a whole bunch of properties. It's going to say title equals self.title, and then author equals, and then I'll just print the author information, and then price. All right, and now let's add a couple of more function calls to convert the book objects to strings by using str and repr directly. So for the first example, I'm actually going to call the str function on b1, and then I'll use repr on b2. All right, so let's go ahead and save, and now let's run the code again. All right, so now you can see that when I print the objects or call str directly, the str function gets used. And when I call the repr function, that causes my double underscore version of repr to be used instead. So each of these functions is totally optional for you to override, but it's usually a pretty good idea to at least define the repr function for classes that you create in order to make debugging easier.

### Equality and comparison

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Plain objects in Python, by default, don't know how to compare themselves to each other. But we can teach them how to do so by using the equality and comparison magic methods. So, let's go ahead and open up the magiceq\_start file. And let's see how to do this. So, once again, we have a book class defined with some attributes, and then a few variables that create some book objects. So b one and b three, both contain the same information. But watch what happens when I try to compare them to each other. So, I'm going to write print, and then b one double equals b three. Now, when I run this, you can see that the output is false, which is weird, because all the attribute values are the same as each other. So, the title is the same, the prices is the same, the author is the same. The reason this happens is because Python doesn't do an attribute by attribute comparison on objects. It just compares two different instances to each other and sees that they're not the exact same object in memory. And therefore it says, oh false, they're not equal to each other. So, Python's flexibility gives us an easy object-oriented way of addressing this problem. The magic method named eq, gets called on your object when it is compared to another object. So, let's go ahead and implement that. And override the eq method, And that takes my object as well as the objects it's being compared to. So, to see if two books are equal, we can just compare the attributes of each one. So, I'm going to return if self dot title is equal to value title, and self dot author is equal to the other objects, author and the price is equal to the other objects price. Okay. We should also make sure that we throw an exception, if we're past an object that's not a book to compare against. So, let me just do that right now. So, I'm going to say if not, is instance, and I'm going to take the value that were passed and compare it to the book class. And if it's not, I'm going to raise a value error that says can't compare book to a non-book. Alright, so now we have that code in place, we can perform the comparison again. And let's add one that we also know is false. So, we'll print b one is equal to b two, and those attributes are different. So, the first one should be true and the second one should be false. So, let's go ahead and run. Alright, sure enough, we have true and we have false. So, what we've essentially done is add the equality check behavior to our book object. And if I try to compare a book to something else, if I say print b one, is equal to 42. Let's run that. And you can see that I'm raising a value error here because I can't compare a book to something that's not a book. Alright, so we can also perform other kinds of comparisons by overriding the corresponding magic method. So, let's add the ability to perform comparisons to our book class. Suppose we wanted to be able to perform a greater than or equal to operation like this. We want to be able to say b two is greater than or equal to b one. Or suppose we wanted to be able to do a less than comparison. Suppose we wanted to be able to say, hey, is b two less than b one. So, there are magic methods that correspond to all the different kinds of logical operators. Greater than, less than, greater than or equal to, so on. Now, that's a lot of methods. So, I'm not going to demonstrate all of them. But let's go ahead and add support for both of these. So, I'll scroll back up, and I'll override the greater than or equal to function. And that takes my object and the comparison one. And then once again, we'll just copy to make sure that we're comparing to a book. And what we'll do here is we'll just return self dot price is greater than or equal to value price. So, let's do a comparison based upon price alone. And then let's do the same thing for less than, so I'll just go ahead and copy this code and paste it in down here, change this to lt. And then once again, I'll change this operator to less than. Okay, so now let's run that code. And let's comment out our prior example for the equality check. So, I'm going to run this. Alright, and we can see that b two's price is 2995, and that is not greater than b one's price which is 3995, that evaluates to false. And b two is less than b one based upon price. So, now we have the ability to do comparisons of book objects. And what's really neat about this is that, now that we have added the less than support, we automatically gained the ability to have our books be sortable. So, let's go back to the code, and let's make a quick list of our books in some random order that we know is not sorted. Alright, so the built-in sort function, uses the less than operator to perform sorting. So, now we can do this, we can write books dot sort, and then I can print out each book title in the newly sorted list. And I'll use a comprehension for this. So, I'll print book title for book in books. Alright, so let's go ahead and comment that out. And now when I run this code, we can see that the books are now all sorted in order from low to high, based on price. And like I said, there are a lot of these methods you can implement in your base classes, and they're documented in Python's data model. And that's this right here. So, the documentation for the data model, contains all of these methods. If you just go ahead and click on the Special method names, heading over here, if you scroll down, you'll see a lot of these special magic method names. These are all the comparison ones right here. So, just go ahead and click on that Special names link in the sidebar, and you can read through these at your own pace.

### Attribute access

Selecting transcript lines in this section will navigate to timestamp in the video

- Python's magic methods also give you complete control over how an object's attributes are accessed. Your class can define methods that intercept the process any time an attribute is set or retrieved. So let's open up the magicattr\_start file and we'll see how this works. So here in my code, I have the Book class which defines some attributes and also overrides the \_\_str\_\_ function that we saw earlier in the chapter to print out a nice version of the object. So let's start off by seeing how we can control access when an attribute's value is retrieved. So Python lets us define a magic method called \_\_getattribute\_\_ which is called whenever the value of an attribute is accessed. So what I'm going to do is override this function. I want \_\_getattribute\_\_. And that is going to be passed, my object and the name of the attribute. So this gives us the opportunity to perform any operations on the value before it gets returned. So for example, you can see that we have an internal attribute named discount. So let's imagine that we wanted to automatically apply the discount whenever the price attribute is retrieved. So first, we can check to see if the name argument is equal to price, so we know that that's the price attribute being accessed. And now, here's where things get tricky. So since we're already inside the function that's going to get called whenever an attribute value gets accessed, we can't just refer to this object's attributes by name because then this function will just get recursively called over and over again and it will eventually crash. So what we need to do is get the value of the current price by calling the superclass version of \_\_getattribute\_\_. So I'm going to write p =, and then I'm going to call the superclass' \_\_getattribute\_\_, and I'll ask for the price, and then I'll do the same thing for the discount, and I'll name that d, and we'll put in \_discount And then, I'll just calculate the discount. So I'll return p minus p times d. And if we're not operating on the price attribute, we'll just go ahead and call the superclass' \_\_getattribute\_\_ so everything just works normally. So I'll just call \_\_getattribute\_\_ with the name. All right, so let's scroll down and you can see I've got some code that creates a couple of Book objects. Let's go ahead and try setting the price of book one to 38.95, and then let's go ahead and print b1. And remember, the print statement will trigger the \_\_str\_\_ output. So I'll run this. And we can see, in the output, that the price of the book is reduced by 10% but, clearly, the other attributes, the title and the author are unaffected. So we can also control the setting of an attribute by overriding the \_\_setattr\_\_ function, so let's try that next. So here, I'm going to override \_\_setattr\_\_, and that takes a reference to my object, the name of the attribute and the new value. So in this case, let's use \_\_setattr\_\_ to enforce that when the price attribute is set, the caller is using a floating point number. So once again, we'll check to see if name is equal to price, and if it is, we'll check to see if the type of the value that we were passed is not float. Then, we're going to raise an error. And we'll just say, the price attr must be a float. Okay, so now we're going to raise an error if the value that we were passed is not a floating point number. So if we pass that test, then we'll just call the superclass' version of \_\_setattr\_\_. So we'll just go ahead and return. And once again, we'll call the superclass, and we'll pass in the name and the value. All right, so once again, let's try this out. I will comment out my previous example, and I'll set the price of book two to be 40, as an integer, so that should cause an error, and then, we'll print b2. So I'm going to save and I'm going to run this. And when I run it, you can see that I'm getting my error right here, it says the price attr must be a float. So let's go back to the code. Now, I could either fix this by making this by making this a floating point number or I can cast it to a floating point number by using the float function. All right, let's save and run it again. And now you can see it works fine, right? So I set it to be 40, and then 10% off of 40 is 36. All right, let's do one more example. Let's go back to the code. There's another magic method that lets us customize the retrieval of attributes, but it only gets called if the given attribute doesn't exist. So it's the \_\_getattr\_\_ version of \_\_getattribute\_\_. So let's go ahead and give that a try. So I'll define that method. And that's going to take self and name. All right, so this version of the function only gets called if the \_\_getattribute\_\_ version either doesn't exist, or if it throws an exception, or if the attribute doesn't actually exist. So let's go ahead and comment out my overriding of \_\_getattribute\_\_. And then, let's implement this version. So all we're going to do is return the name and we're going to say, is not here. All right, and then, let's try to access an attribute that we know doesn't exist. So I'll comment this out and I'll just try to print out, you know, b1., and I'll call it randomprop. All right, so let's go ahead and run this. And you can see that now I'm getting the output, randomprop is not here. Now, you could use this to generate attributes on the fly, or if you wanted to extend the syntax for accessing attributes, but you know, just like other methods, you need to be careful that you don't enter into a recursive loop. But by using these attribute methods, Python gives you a great amount of flexibility and control over how attributes are retrieved and set in your classes.

### Callable objects

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] To finish up this chapter, we'll take a look at the magic method that enables Python objects to be callable just like any other function. Now that might sound a little bit weird, but it's easy to understand when you see it in action. So let's go ahead and open up the magiccall\_start file, and once again, you can see that I have my book class, and it's already implementing the str magic function that learned about earlier. And I've got some code that creates a couple of book instances with titles, and authors, and prices. So what I'm going to do is this, I'm going to override the call function that lets me treat the instance of the object like a function. And I'll define the function to take the same parameters as the init function. Now you can also define the function to take a variable number of arguments, but that's a little more advanced, and I want to focus here on the feature itself. So I'm going to define call, and this method will be invoked when I call my object. So it gets a copy of the object, and then I'll pass in the title, the author, and the price, and then for the function body, I'll just assign the parameters to the object attributes, pretty much just like I have here in my init function. So I'll just go ahead and do that. Okay, so now let's try this out. So first what I'm going to do is print book1's values, then I'm going to call the object like a function to change the values of the object's attributes. So I'll write b1, and then I'm going to call it, just like a function. And this time I'll change the title to something else, and then author's going to stay the same, and the price is going to change, and then I'll print the book again, and remember, this will use the str method to do so. Okay. So let's try this out. And when I run the code, you can see here that I'm changing the value of the object's attributes by calling the object as if it were a function. So here's the original version, and here is the version after I've changed the values. And that's one of the benefits of this technique is if you have objects whose attributes change frequently, or are often modified together, this can result in more compact code that's also easier to read.

### Defining a data class

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] As we've been working through this course, you may have noticed a pattern with each of our examples so far. And that is that one of the main use cases for creating classes in Python is to contain and represent data. Our code creates classes like this book class here in the data class start file and then uses the init function to store values on the instance of the class. And you might be wondering, well, if this is such a common pattern, then why doesn't Python just automate this? Why do I have to explicitly store each argument on the object by setting attributes on the self parameter? Well, starting in Python 3.7, you actually don't. In 3.7, Python introduced a new feature called the data class, which helps to automate the creation and managing of classes that mostly exist just to hold data. And that's what we're going to focus on in this chapter. You can read more about data classes at this link in the Python docs. But for now, let's go back to our code, and let's begin by converting our book class into a version that uses a data class. So to do that, the first thing we need to do is import the data class from the data classes module. So from dataclasses we're going to import dataclass. And again, this only works in 3.7 and later, so make sure you have at least that version of Python on your computer. Next, we're going to use the dataclass decorator to indicate that the book class is going to be a dataclass. Then, we get rid of the init function, and let's go ahead and fix the indenting here. And we're going to get rid of each of these self keywords, because there's no more self keyword right now. And then we need to annotate each of these attributes with the new type hints that were introduced in Python 3.5. So I'm going to get rid of each of the equal signs, I'm going to write title, and then colon str. And then the author is also going to be a string, so that's a str. Pages is going to be an int. And price is going to be a float. And guess what? That's pretty much all we have to do. Now, there's quite a bit going on here, so let me explain. At first glance, it looks like what we're doing is defining class attributes instead of instance attributes. But what's going to happen behind the scenes is that the dataclass decorator code will actually rewrite this class to automatically add the init function where each of these attributes will be initialized on the object instance. And the second thing you notice here is these type hints. These are required for data classes to work. But in keeping with Python's philosophy of being flexible, their type isn't actually enforced. So you can see here, we have some existing code that creates some book instances and then accesses some fields. And we don't even need to change our existing code as long as the parameters are passed in the same order and they are. We've got title, author, pages, and price. So everything looks fine. So let's go ahead and run what we have. I'm going to save and then run this. So here in the output you can see we've got the title and the author of what is that, book one and book two, right. So, we can access attributes on the object just as we could before. But dataclasses have more benefits than just concise code. They also automatically implement both the repr and eq magic methods we learned about earlier in the course. So, for example, I can add a print statement here to just print out book one. And we can compare two objects with each other, so I'm going to make another book that has the same values as book one. And I'll call that book three. And then I'll add a comparison to see if book one is equal to book three. All right, so, now let's go ahead and run again. And you can see that the printed output of the book object automatically contains all of the data attributes and the equality comparison also just automatically works. All right, so one more thing to demonstrate for our intro to data classes. So they are just like any other Python classes. There's nothing really special about them. They're a Python class just like any other Python class. So if I want to add a regular Python method to my class, it's really straightforward to do so. So I can just go ahead and create a method called bookinfo, and I'll just return some formatted string that contains self.title and self.author. And then let's go ahead and modify some attributes of one of the books. And then let's go ahead and call that method, b1.bookinfo. All right, so let's go ahead and comment out some of these others. Okay, so let's go ahead and run this. And there you can see that our book info method works just as expected. So data classes let you write a lot more concise code and skip a lot of the boilerplate that comes along with the init method and initializing object instances but at the same time, they're just regular Python classes and you can use them just as you would any other Python class.

### Using post initialization

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] All right, let's continue looking at data classes by looking at the post\_init function. So at this point, you might be wondering how you can perform additional object initialization if the data class automatically writes the init function for you. So for example, we might want to create attributes on our book class that depend on the values of other attributes, but we can't write init because the data class is going to do that, so what do we do? So to accomplish this, the data class decorator provides a special function named post\_init that you can override and that will be called for you after the built-in init function has finished. So here in the code of the post\_init start file, let's go ahead and add an attribute that combines the title and the author strings. So all I need to do is override the post\_init function and I'm going to override it just like I would any other Python function, and that takes a reference to my object instance. And then let's make an attribute that combines the title and author values. So I'll write self.description is equal to, and I'll use a formatted string for this, and I'll just combine the self title by self.author, and it's self.pages pages. All right, so when the data class gets finished initializing the main attributes here, which we've already listed, this function is going to get called, and any additional attributes we want to create can be done so in post\_init. So let's go ahead and add some code to try out this new attribute. So we create the books, and then we're going to print out b1.description, and I'm also going to print out b2.description. All right, so let's go ahead and save, and we'll run our code. And there you can see, we have the new description attribute which combines both the title and the author. So the post\_init function is where you can add any additional attributes that might depend on the values of other attributes in your object when you're using data classes.

### Using default values

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Dataclasses provide the ability to define default values for their attributes subject to some rules. So let's go ahead and open up datadefault\_start. And let's try this out. So you can specify default values for attributes, just as if you had declared them in the init function for the class. So let's go ahead and add some default values to our Book class. So the title will default to a string that says No Title. And we'll have the default for author be No Author. And pages and price will both default to zero and zero. Okay, so then I can create a book with no arguments. And then I can print it out. All right, so let's go and run this. We can see the output that the book object was created with all of the default values that we specified in the dataclass definitions. Now, one of the things you need to be aware of with default values is that attributes without default values have to come first. So let's go ahead and take off the default value from the float, all right? And then save. Now if I try to run this again, you'll see that in the output, I get an error. And it says right here, TypeError that a non-default argument price follows a default argument, all right? So let's go ahead and put that default back. Another way of defining a default value is by using the field function, which provides a little more flexibility. So first, I'm going to import the field class from the dataclass module. And then I can use the field function to specify a default value. So instead of having a constant there, I'm going to write field and then put in default equals and then the value. So now let's add some code to create a couple of books with some non-default data except for the price. And then we'll print out books b1 and b2. All right, so now let's run the code again. And actually, you know what, instead of having it be zero, I'm going to set it to be 10, just so we can see it stand out. Let's go ahead and run this again. And you can see that in the result, the price default is now 10. But we can do a little bit better than that. We can also specify a function that gets called in order to provide the default value. So I'll make a slight modification. I'm going to change the argument to the field function from default to default factory, which will call a function to provide the default value. So let's go back up into my code. And I'm going to change this to read default\_factory. And we're going to have this call a function named price\_func. And I'm going to write a function that will generate a random price between 20 and 40. So that means I'm going to need a random number. So I'm going to import the random module. And then I'm going to write the price function. And I'll just do that outside the dataclass. So I'll write price\_func. And that's going to return a floating point number and I'm going to call random.randomrange for between 20 and 40. All right, so you can see that there are various ways to provide default values. Let's go ahead and run this again. So now you can see that the prices are being set to these random numbers, we have 36 and 35. So let's go ahead and run it again. And now we got 21 and 31. One more time, all right, 27 and 39. So you can see that there are various ways to provide default values for dataclass attributes. You can specify them directly in the attribute list, you can use the field function, or you can call a function that can generate a value dynamically. Just remember that default values have to come first. You can't have non-default values following default values.

### Immutable data classes

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Occasionally you'll want to create classes whose data can't be changed. In other words, you want the data within them to be immutable. Python data classes make this possible by specifying an argument to the data class decorator. So let's go ahead and open up the immutable\_start file, and you can see here I have a simple data class definition with a couple of attributes, along with some code that creates the object and prints out an attribute value. So let's go ahead and runt his to make sure it works. And in the output you can see that it works, my class gets created, and then value1 gets printed out. So let's go ahead back to the code. To make this class immutable, I can set the frozen argument to true in the data class decorator. So I'll just simply write frozen equals true. So this now prevents any of the attributes in the class definition from being modified. So I'll try to add some code now to do exactly that. So I'll go ahead and put obj1, and then I'll try to change the value of the value1 attribute to another value. And then we'll print that out. So let's go ahead and try to run the code now. And when I try to run the code, you can see that I get a frozen instance error, and you can see the explanation says cannot assign value to field value1. So this also prevents modification within the class itself, not just from outside the class. Let's go ahead and add a function to the class, and I'll define a method named somefunc, and then I'll take the object and a new value, and we'll set some new attribute value on value2. All right, and then down here we'll try to call the function with a new value. And let's go ahead and comment that out, because we know that that doesn't work. So I'm going to save this, and I'm going to run it once more. And then once again, you can see I'm getting the same frozen instance error. So creating frozen data classes can be useful when you want the class to represent data that you know isn't going to change, and with a data class decorator, that's really easy to do.

### Next steps

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] All right, congratulations on making it to the end of the course. I hope you enjoyed learning about how the Python language uses object-oriented programming techniques and maybe you've picked up some ideas about how to apply these principles in your own Python programs. Now as I said at the beginning of the course, it's not required to use object-oriented programming with Python, but I think that once you get started with it and you've experienced the benefits, you'll be reluctant to go back to writing code the old non-object-based way. There are plenty of other Python-focused courses to explore while you're here. So I suggest checking a few of them out. I hope to see you again soon in another one of my courses. Until then, happy coding!