**Course Overview**

**Course Overview**

**Hi everyone. My name is Robert Smallshire. Welcome to our course, Advanced Python. I'm a principle consultant at Sixty North, and out of the dozen or so programming languages I can choose to use, Python is the one I return to again and again to produce low friction solutions quickly and efficiently. I've been working regularly with Python since the turn of the millennium, using it to solve problems across a wide range of business, scientific, and engineering sectors ranging from simulating geological processes in planet Earth through to monitoring and controlling electricity consumption in industrial plants over the web. Over that time Python has exploded in popularity from being an obscure scripting language to becoming one of the most popular and widely used language in the world. The growing demand for Python skills reflects it's broad adoption in the diverse realms of web development, data science, cloud operations, and as an embedded scripting language in professional applications. This course is the culmination of our series on the core Python language. We'll cover advanced topics, a knowledge of which will set you apart from the greater number of Python developers. Topics, such as descriptors, we'll learn how to gain fine grain control over attribute access, metaclasses, where you'll see how to intercept class object construction, and virtual subclasses, where you'll discover the subtle but powerful controls Python gives you over class relationships. By the end of this course you'll know enough Python to understand the advanced techniques used to implement sophisticated frameworks, like SQLAlchemy or Django, and be able to bring those capabilities to bear in your own solutions. Before beginning this course you should be familiar with the material we cover in our Python Fundamentals and Python Beyond the Basics courses here on Pluralsight. I hope you'll join me on this journey to expand your core Python 3 language skills with the advanced Python course at Pluralsight.**

**Advanced Flow Control**

**Advanced Python**

**Welcome to the Advanced Python course. My name is Robert Smallshire. Advanced Python is the third in our trilogy of 3 courses, which cover the core Python language, and it builds directly on the knowledge we impart in our Python Beyond the Basics and Python Fundamentals courses, which comprise the second and first parts of our core Python 3 training. Our courses follow a thoughtfully designed spiral curriculum. We visit the same or closely related topics several times in increasing depth, sometimes multiple times in the same course. For example, in Python Fundamentals we covered single class inheritance. In Python Beyond the Basics we covered multiple class inheritance. In this advanced Python course we'll cover what amounts to a type of virtual inheritance using Python's sophisticated abstract base class mechanism. In Python Fundamentals we covered basic techniques you'll use constantly. In this advanced Python course we cover some aspects of Python, which you may use only occasionally. Nevertheless, mastery of the Python language calls for felicity with these features, at least to the extent that you can identify their use in existing code and appreciate their intent. We'll go all the way in this course to show you how to use the most powerful of Python's language features to greatest effect with occasional reminders that sometimes, though, there's a simpler way. Knowing how to use advanced features is what we'll teach in this course. Knowing when to use advanced features, demonstrating a level of skillfulness that can only be achieved through time and experience is the mark of the true master. Specifically in this class, we'll cover advanced flow control, including loop else clauses, try else, and switch emulation. Then we'll go on to examine byte-oriented programming where we'll demonstrate how to interpret and manipulate data at the lowest level in Python. We'll open up objects to inspect their insides where we'll see how objects are represented internally in Python. Then we'll introduce descriptors, showing how to gain complete control over attribute access using a crucial mechanism in Python, which is usually behind the scenes. We'll show how to customize object allocation to make the most efficient use of memory with tactics like interning of immutable objects. That will support our exploration into the process by which class declarations are transformed into class objects by metaclasses, and demonstrate how to write your own metaclasses to customize class construction. Remembering that sometimes there's a simpler way, we'll explain how class decorators can be a simpler, although less powerful, alternative to metaclasses in many cases. We'll finish off this course by showing you how to use abstract base classes to specify and detect class interface protocols, and even how you can make built-in types become subclasses of your own types. That's a lot to cover, but as we go through this course you'll begin to see how many of these pieces fit together. Throughout this course we'll expect you to be familiar with much of the material covered in our Python Fundamentals and Python Beyond the Basics courses. You'll need to be confident of using the built-in types, defining functions, classes, and modules. You'll need to be comfortable defining and using decorators and constructs such as static methods, class methods, properties, and other approaches we have previously introduced, which will be used without introduction. In other words, we expect you to be comfortable with basic and intermediate Python programming techniques before embarking on this course, whether or not you've experienced our earlier classes. You also need a functioning Python 3. 5 or later development environment. Nothing in our courses depends on specific tools other than a Python interpreter, so if you're trying out our code examples for yourself you can use whichever editor you're most comfortable with. For all the demonstrations in this course, we're using JetBrains PyCharm. We need to make a quick note regarding terminology. In Python many language features are implanted or controlled using special methods on objects. These special methods are generally named with two leading and two following underscores. This has the benefit of making them visually distinct, fairly easy to remember, and unlikely to collide with other names. This scheme has the disadvantage, however, of making these names difficult to pronounce, a problem we face when making courses like this. To resolve this issue we have chosen to use the term "dunder" when referring to these special methods. Dunder is a portmanteau of the term double underscore, and we'll use it to refer to any method with leading and trailing double underscores. For example, when we talk about the method, \_\_len\_\_, which is invoked by the len function, we'll say dunder len. These kinds of methods play a big role in this course, so we'll be using this convention frequently. If you'd like a book to support you as you work through the material in this course you can check out The Python Master, which is the companion volume to this advanced Python course covering the same material in written form. By following the URL shown you can obtain the book for a substantially discounted price. The Python Master is the third book in our Python Craftsman trilogy, the first two books being The Python Apprentice, and The Python Journeyman, which correspond to our Python Fundamentals and Python Beyond the Basics Pluralsight courses, respectfully. All three are available to Pluralsight viewers at reduced prices. Like any advanced activity, it takes time and experience to build the skills and control you need to achieve success. With the basic training done, let's launch ourselves into the adventure of advanced Python programming starting out with the twists and turns of advanced flow control.**

**Introducing Advanced Flow Control**

**In this module we'll be looking at some of the more advanced flow control techniques used in Python programs, with which you should be familiar at an advanced level. Specifically, we'll cover else clauses on loops, else clauses on try blocks, emulating switch statements, and dispatching on type. By understanding these unusual language features you'll be able to understand more code that you may encounter, and reduce the complexity of your own code by eliminating unnecessary conditionals.**

**Loop-else Clauses and While-else**

**You don't disassociate the else keyword with optional clauses complimentary to the conditional clause introduced by the if statement, but did you know that else can also be used to associate optional code blocks with loops? That sounds pretty weird, and to be honest, it is an unusual language feature, which is only rarely seen in the wild, and which most definitely deserves some explanation. Now let's look at the while else construct. We'll start out by saying that Guido van Rossum, inventor and benevolent dictator for life of Python has admitted that he would not include this feature if he developed Python again. Back in our world, though, the feature is there, and we need to understand it. Apparently, the original motivation for using the else keyword this way, in conjunction with while loops comes from Donald Knuth in early efforts to rid structured programming language of go to. Although the choice of keyword is at first perplexing, it is possible to rationalize the use of else in this way if your comparison with the if else construct. In the if else structure the code in the if clause is executed if the condition evaluates to true when converted to Bool. In other words, if the condition is truthy. On the other hand, if the condition is falsy the code in the else clause is executed instead. Now look at the while else construct. We can, perhaps, glimpse the logic behind choosing the else keyword. The else clause will be executed when and only when the condition evaluates to false. Of course, the condition may already be false when execution first reaches the while statement, so may it may branch immediately into the else clause or there may be any number of cycles of the while loop before the condition becomes false and execution transfers to the else block. Fair enough, you might say, but isn't this the equivalent of just putting the else code after the loop like this rather than in a special else block? You would be right in this simple case, but if we place a break statement within the loop body it becomes possible to exit the loop without the loop conditional ever becoming false, so now the execute condition is false function call happens even though the condition is not false. To fix this we could use a second test with an if statement after the loop to provide the correct behavior. The drawback with this approach is that the test is duplicated, which violates the Don't Repeat Yourself or DRY guideline, which hampers maintainability. The while else construct in Python allows us to avoid this second redundant test. Now the else block only executes when the main loop condition evaluates to false. If we jump out of the loop another way, such as with the break statement, execution jumps over the else clause. There's no doubt that, however you rationalize it, the choice of keyword here is confusing, and it would have been better all-around if a no break keyword had been used to introduce this block. In lieu of such a keyword, we heartily recommend that if you are tempted to use this obscure and little used language feature that you include just such a nobreak comment, like this.**

**While-else for Evaluating Stack Programs**

**So much for the theory. Is this any use in practice? Well, we must admit that neither of the authors of this course have ever used while else in practice. Almost every example we've seen could be implemented better by another more easily understood construct, which we'll look at later. In this demo our code evaluates simple stack programs where a program is specified as a stack of items where each item is either a callable function, for these we just use any regular Python function, or an argument to that function. So to evaluate 5 + 2 we would set up a stack like this with plus placed first onto the stack, then 2 and then 5. When the plus operator is evaluated its result is pushed onto the stack. This allows us to perform more complex operations, such as (5+2)\*3 by pushing multiply onto the stack, then 3, then plus, then 2, and then 5. When evaluating the stack we collect operands until we reach an operator, so we collect 5 and 2, then pop the operator, plus, from the stack, replacing it with 5+2 or 7. Continuing, we pop the operands 7 and 3 from the stack until we reach the operator, multiply. We then push 7 multiplied by 3 back onto the stack, which is 21, the final result. As the stack contains the expression in reverse polish notation the parentheses we needed in the infix version aren't required. In reality, the stack will be a Python list, and the operators will be callables from the Python standard library operator's module, which provides named function equivalents of every python infix operator. It's important to note that when we use Python lists as stacks the top of the stack is the end of the list, so to get everything in the right order we need to reverse our list using the reversed built-in function. For added interest our little stack language also supports comments of strings beginning with a hash symbol, just like Python; however, such comments are only allowed at the beginning of the program, which is at the top of the stack. We'd like to run our little stack program by passing it to a function execute like this. Let's see what such a function might look like, and how it can use the while else construct to good effect. The first thing our execute function needs to do is pop all the comment strings from the top of the stack and discard them. To help with this we'll define a simple predicate function, which identifies stack items as comments, called is\_comment. Notice that this function relies on an important Python feature called Boolean Short Circuiting. If item is not an instance of str, then the call to the startswith method will cause an attribute error to be raised; however, when evaluating the Boolean operators and and or Python will only evaluate the second operand if it is necessary to compute the result. In the case that the item is not a string, and the first operand evaluates to false, the results of the Boolean, and, must also be false, with no need to evaluate the second operand. Given this useful predicate, we'll now use a while loop to clear comment items from the top of the stack. The conditional expression for the while statement is the program stack object itself. Remember that using a collection in a Boolean context like this evaluates to true if the collection is non-empty or false if it is empty, or put another way, empty collections are falsy. So this statement reads, while there are items remaining in the program. Within the while block we pop an item from the stack, recall that regular Python lists have this method, which removes and returns the last item from a list. We use logical negation of the results of our is\_comment predicate to determine if the just popped item is not a comment. If the loop has reached a non-comment item we push it back onto the stack using a call to append, which leaves the stack with the first non-comment item on top, and then break from the loop. The while loop has an associated else clause to where execution will jump if the while condition should ever evaluate to false, which is when there are no more items remaining in the program. In this clause we print a warning that the program is found to be logically empty, and return early from the execute function. Remember that the while loop else clause is best thought of as a nobreak clause, so when we break from the loop execution skips the else block and proceeds with the first statement after. This loop executes the else block in the case of a search failure. We failed to locate the first non-comment item because there wasn't one. Search failure handling is perhaps the most widespread use of loop else clauses. Now we know that all the remaining items on the stack comprise the actual program. We'll use another while loop to evaluate it. Before the loop we set up an empty list called pending. This will be used to accumulate arguments to function calls, as we'll see shortly. As before, the condition on the while loop is the program stack itself, so this loop will complete and control will be transferred to the while loop else clause when the program stack is empty, which happens when program execution is complete. Within the while loop we pop the top item from the stack, and inspect it with the built-in callable predicate to determine if it is a function. For clarity, we'll look at the else clause first. That's the else clause associated with the if not the else clause associated with the while. If the popped item is not callable we append it to the pending list, and go around the loop again if the program is not yet empty. If the item is callable we attempt to call it, passing any pending arguments to the function using the \*args construct. Should the function call succeed, we'll assign the return value to result, push this value back onto the program stack, and clear the list of pending arguments. Should the function call fail, we'll catch the exception, print an error message, and break from the while loop. Remember that this will bypass the loop else clause. When the program stack is empty the else block associated with the while loop is entered. This prints Program successful, followed by any contents of the pending list. This way the program can return a result by leaving non-callable values at the bottom of the stack. These will be swept up into the pending list, and displayed at the end.**

**For-else Clauses and Handling Search Failure**

**Now we understand the while else construct, we can look at the analogies for else construct. This may seem even more odd, given the absence of an explicit condition in the for statement, but you just need to remember that the else clause is really the nobreak clause, and in the case of the for loop, that's exactly when it was called. This includes the case when the iterable series over which the loop is iterating is empty. As such, the else clause is useful to handle the not found case when searching for something. The typical pattern of use is like this. We use a for loop to examine each item over an iterable series, and test each item with an if statement. If the item matches we break from the loop. In the event that we fail to find a match, the code in the else block is executed, which handles the No match found case. In the next demo we'll use a for else loop to ensure that a sequence contains at least one integer divisible by a specified value. For example, are any of 2, 5, 9, 37, 28, or 14 divisible by 12? No, so let's append a number that is, 12. Here's a code fragment which ensures that a list of integers contains at least one integer divisible by a specified value. In the event that the supplied list does not contain a multiple of the divisor, the divisor itself is appended to the list to establish the invariant. We set up a list of numeric items and a divisor, which will be 12 in this case. Our for loop iterates through the items, testing each in turn for divisibility by the devisor. If a multiple of the divisor is located the variable it found is set to the current item, and we'll break from the loop, skipping over the loop else clause, and printing the list of items. Should the for loop complete without encountering a multiple of 12, the loop else clause will be entered, which appends the divisor itself to the list, thereby ensuring that the list contains an item divisible by the divisor. For else clauses seem to be much more common than while else clauses, although we must emphasize that neither are common, and both are widely misunderstood, so although we want you to understand them, we can really recommend using them widely unless you assure that everyone who needs to read your code is familiar with their use. In a survey undertaken at PyCon 2011 a majority of those interviewed could not properly understand code which used loop else clauses. Proceed with caution.**

**Alternatives to Loop-else Clauses**

**Having pointed out that loop else clauses are best avoided, it's only fair that we provide you with an alternative technique, which we think is better for several reasons, beyond just avoiding an obscure Python construct. Almost any time you see a loop else clause you can refactor by extracting the loop into a named function (Typing), and instead of breaking from the loop simply return directly from the function. The search failure part of the code, which was in the else clause, can then be dedented a level, and just be placed after the loop body. Doing so our new ensure\_has\_divisible function would look something like this. This is simple enough to be understood by any Python programmer. We can use it like this, passing the items and the divisor, and accepting the dividend as a result. This is easier to understand because it doesn't use any obscure and advanced Python flow control techniques. It's easier to test because it's extracted into a standalone function. It's reusable because it's not mixed in with other code, and we can give it a useful and meaningful name rather than having to put a comment in our code to explain the block. Better all around.**

**Try-else Clauses**

**The third slightly oddball place we can use else blocks is as part of the try except exception handling structure. In this case, the else clause is executed if no exception is raised. Looking at this, we might wonder why we don't just call do\_something\_else on the line after do\_something, like this. The down side of this approach is that we now have no way of telling in the except block whether it was do\_something or do\_something\_else, which raised the exception. The enlarged scope of the try block also obscures our intent with catching the exception. Where were we expecting the exception to come from? Although rarely seen in the wild, it is quite useful, particularly when you have a series of operations which may raise the same exception type, but where you only want to handle exceptions from the first such operation. In this example, both opening the file and iterating over the file can raise an OS error, but we're only interested in handling the exception from the call to open. Note that it's possible to have an else clause and a finally clause. The else block will only be executed if there was no exception, whereas the finally clause will always be executed.**

**Emulating Switch Statements**

**Most imperative programming languages include a switch or a case statement, which implements a multi-way branch based upon the value of an expression. Here's an example for the C programming language where different functions are executed depending on the value of the menu\_option variable. There's also a default handler for no such option. Although switch can be emulated in Python by a chain of if elif else blocks this can be tedious to write and is error prone because the condition must be repeated multiple times. An alternative in Python is to use a mapping of callables. Depending on what you want to achieve, these callables may be lambdas or named functions. We'll look at a simple adventure game you cannot win in Kafka. py, which will refactor from using if elif else blocks to using dictionaries of callables. Along the way we'll also use try else. The game loop uses two if elif else chains. The first prints information dependent on the players current position, then after accepting a command from the player, the second if elif else change takes action based upon the command. Let's refactor this code to avoid those long if elif else chains, both of which feature repeated comparisons of the same variable against different values. The first chain describes our current location. Fortunately, in Python 3, although not in Python 2, print is a function, and can therefore be used in an expression. We'll leverage this to build a mapping of position to callables called locations. We wrap each call to print in a 0 argument lambda to make it a callable. We'll loop up one of these callables using our position in locations as a key, and call the resulting callable in a try block. Notice here that we don't really intend to be catching KeyErrors from the callable, only from the dictionary lookup, so this also gives us an opportunity to narrow the scope of the try block using the try else construct we learned about earlier. We separate the lookup and the call into separate statements and move the call into the else block. Similarly, we can refactor the if elif else chain, which handles user input into a dictionary lookup for a callable, although this time we used name functions rather than lambdas to avoid the restriction that lambdas can only contain expressions and not statements. Again, we split the lookup of the command action from the call to the command action using the try else construct. Here are the five callables referred to in the dictionary values, go\_north, go\_east, go\_south, and go\_west, look, and quit. Notice that using this technique forces us into a more functional style of programming. Not only is our code broken down into many more functions, but the bodies of those functions can't modify the state of the position variable. Instead, we pass in this value explicitly, and return the new value. In the new version mutation of this variable only happens in one place rather than five. Although the new version is larger overall, we'd also claim it's much more maintainable. For example, if a new piece of game state, such as the players inventory were to be added, all command actions would be required to accept and return this value, making it much harder to forget to update the state than it would be in chained if elif else blocks. Let's add a new rabbit hole location, which when the user unwittingly moves into it, leads back to the starting position of the game. To make such a change we need to change all of our callables in the location mapping to accept and return a position and aliveness status. We must also update the call to location\_action to pass the current state and receive the modified state. Here are the new location functions; labyrinth, dark\_forest\_road, tall\_tower, and rabbit\_hole. Although this may see onerous it's a good thing. Anyone maintaining the code for a particular location can now see what state needs to be maintained. Now let's make the game a little more morbid by adding a deadly lava\_pit location, which returns false for the alive status. Here's the function for the lava pit location, which simply prints, you fall into a lava pit, and then returns the current position, but false for aliveness, and of course, we must remember to add this to the location dictionary. We'll also add an extra conditional block after we visit the location to deal with deadly situations, printing, you're dead, if we're not alive, and then breaking from the game loop. Now when we die by falling into the lava pit we break out of the while loop. This gives us an opportunity to use a while else clause to handle non-lethal came loop exits, such as choosing to exit the game, which sets the position variable to non, which is falsy. Now when we quit deliberately, setting position to non, and causing the while loop to terminate we see the message from the else block associated with the while loop. You have chosen to leave the game. You are in a maze of twisty passages, all alike. Okay. Let's go east. You are on a road in a dark forest. To the north you can see a tower. Right, the tower sounds interesting. Let's go north. There is a tall tower here, with no obvious door. A path leads east. Hmm. This game isn't so interesting, let's quit. You have chosen to leave the game. Game over. But when we die by falling into the lava pit, which causes alive to be set to false, causing execution to break from the loop we don't see the message, as the else block is skipped. You are in a maze of twisty passages, all alike. Twisty passages again, ahh, we'll still go east. You are on a road in a dark forest. To the north you can see a tower. I'm still intrigued by that tower. Let's go north again. There is a tall tower here with no obvious door. A path leads east. Hmm, a door. Let's try north. You fall into a lava pit. You're dead. Game over.**

**Dispatching on Type**

**That was far too much excitement. Let's look at dispatching on type. To dispatch on type means that the particular implementation of a function, which will be executed, depends, in some way, on the arguments to that function. Python dispatches on type whenever we call a method on an object. There may be several implementations of that method in different classes, and the one that is selected depends on the type of the object. Ordinarily, we can't use this sort of polymorphism with regular functions, and we need to resort to ungainly switch-emulation to root cost the appropriate implementation. The singledispatch decorator, which we'll introduce in this demo, provides a more elegant solution to this problem. Consider the following code, which implements a simple inheritance hierarchy of shapes, specifically, a circle, a parallelogram, and a triangle, all of which inherit from a single base class called shape. Each class has an initializer and a draw method. The initializers store any geometric information peculiar to that type of shape, and pass further arguments up to the shape base class, which stores a flag indicating whether the shape is solid. When we say shape. draw in the main the particular draw method that is invoked depends on whether shape is an instance of Circle, Parallelogram, or Triangle. Of course, on the receiving end it's clear that the object referred to by shape becomes referred to by the first formal argument to the method which, as we know, is conventionally called self. So we say the call is dispatched to the method depending on the type of the first argument. Of course, our code works as you'd expect, and even our little Unicode shape glyphs display properly. This is all very well, and is the way much object oriented software is constructed, but can lead to poor class design because it violates the single responsibility principle. Drawing isn't a behavior inherent to shapes, still less drawing to a particular type of device. In other words, shape classes should be all about shapeness, not about things you can do with shapes, such as drawing, serializing, or clipping. What we'd like to do is move the responsibilities which aren't intrinsic to shapes out of the shape classes. In our case, our shapes don't do anything else, so they simply become containers of data with no behavior, like this. Now our three concrete shape classes, circle, parallelogram, and triangle have only initializers. With the drawing code removed from the classes there are several ways to accomplish implementing the drawing responsibility outside of the classes. In this version of draw we test shape using up to three calls to is instance against circle, parallelogram, and triangle. If the shape object doesn't match any of those classes we raise a type error. This is all good to maintain, and is rightly considered to be very poor programming style. Another approach is to emulate a switch using a dictionary lookup where the dictionary keys are types, and the dictionary values are the functions which do the drawing. Here we look up a drawer function by obtaining the type of the shape in a try block, translating the key error to a type error if the lookup fails or on the happy path of no exceptions, invoking the drawer with the shape and the else clause. This looks better, but is actually more fragile because we're doing exact type comparisons when we do the key lookup, so a subclass of, say, circle wouldn't result in a call to draw circle. The solution to these problems arrived in Python 3. 4 where we can use the singledispatch decorator defined in the Python standard library functools module to perform dispatch on type. Functions which support multiple implementations, dependent on the type of their arguments, are called generic functions, and each version of the generic function is referred to as an overload of the function. The act of providing another version of a generic function for different argument types is called overloading the function. These terms are common in statically typed languages, such as C#, C++, or Java, which are rarely heard in the context of Python. In earlier versions of Python, including Python 2, you can install the singledispatch package from the Python Package Index. To use singledispatch we simply define a function decorated with the singledispatch decorator. Specifically, we define a particular version of the function, which will be called if a more specific overload has not been provided. We'll come to the overloads in a moment. At the top of the file we need to import singledispatch with from functools import singledispatch, and then lower down we implement the generic draw function. In this case, a generic function will simply raise a type error. Remember, this is the version that we'll recall when we haven't provided an overload for a specific type of shape. Recall that decorators wrap the function to which they replied, and bind the resulting wrapper to the name of the original function, so in this case, the wrapper returned by the decorator is bound to the name draw. The draw wrapper has an attribute called register, which is also a decorator, which can be used to provide additional versions of the original function, which work on different types. This is function overloading. Since our overloads will all be associated with the name of the original function, draw, it doesn't matter what we call the overloads themselves, so by convention we call them underscore, although this is by no means required. Whatever name you use will be ignored. Here's an overload for circle, another for parallelogram, and a third for triangle. By doing this we have cleanly separated concerns. Now drawing is dependent on shapes, but not shapes on drawing. Our main function now looks like this. It just calls the global scope generic draw function for each item, and the single dispatch machinery will select the most specific overload if one exists or fall back to the default implementation. We could add other capabilities to shape in a similar way by defining other generic functions, which behave polymorphically with respect to the shape types.**

**Double Dispatch with Methods**

**You need to take care not to use the singledispatch decorator with methods. To see why consider this attempt to implement a generic intersect predicate method on the circle class, which can be used to determine whether a particular circle intersects instances of any of three defined shapes. We've decorated the intersects method with singledispatch, and in this base version we just raise a type error if the shape is unknown. We then provide three specific overloads registered with intersects, which now had to intersect with other circles, parallelograms, or triangles. Each of these methods in turn delegates to a more specific function, circle\_intersects\_circle, circle\_intersects\_parallelogram, or circle\_intersects\_triangle. At first sight this looks like a reasonable approach, but there are a couple of problems here. The first problem, which has been detected here by PyCharm is that we can't register the type of the class currently being defined with the intersects generic function because we haven't yet finished defining it. In this case, we can't register circle because we're in the class definition for circle. The second problem is more fundamental. Recall that singledispatch dispatch is based only on the type of the first argument. When we're calling our new method like this, do\_intersect=my\_circle. intersects(my\_parallelogram), it's easy to forget that my\_parallelogram is actually the second argument to circle. intersects, and that my\_circle is the first actual argument, which becomes the first formal argument called self. Because self will always be a circle in this case, our intersect core will always dispatch to the first overload irrespective of the type of the second argument. This behavior prevents the use of single dispatchery of methods. All is not lost, however. The solution is to move the generic function out of the class, and to call it from a regular method, which simply swaps the arguments. We move the generic function, intersects, out to global scope, and rename it to intersects\_with\_circle. The replacement intersects method of circle, which accepts the formal arguments self and shape, now delegates to intersects with circle with the actual arguments swapped to shape and self. To complete this example we would need to implement two other generic functions, intersects\_with\_parallelogram and intersects\_with\_triangle, although we'll leave that as an exercise. Doing so would give us a complete implementation of not just single dispatch, but double dispatch, allowing us to do shape. intersects(other\_shape) where the function called is selected based upon the types of both shape and other\_shape without the shape classes themselves having any knowledge of each other, keeping coupling in the system manageable.**

**Summary**

**Well, that just about wraps up this module on advanced flow control in Python 3. Let's summarize what we've learned. We looked at else clauses on while loops, drawing an analogy with a much more well-known association between if an else. We showed how the else block is executed only when the while loop condition evaluates to false. If the loop is executed by other means, such as via a break or a return, the else clause is not executed. As such, else clauses on while loops are only ever useful if the loop contains a break statement somewhere within it. The loop else clause is a somewhat obscure and little used construct. We strongly advice commenting the else keyword with a nobreak remark, so it's clear under what conditions the block is executed. The related for else clause works in an identical way. The else clause is effectively the no-break clause, and is only useful if the loop contains a break statement. They are most useful with for loops when searching. When an item is found while iterating we break from the loop, skipping the else clause. If no items are found, and the loop complete naturally, without breaking, the else clause is executed, and code handling the not-found condition can be implemented. Many uses of loop-else clauses, particularly for searching, may be better handled by extracting the loop into its own function from which execution is returned directly when an item is found, and then the code after the loop can handle the not found case. This is less obscure, more modular, more reusable, and more testable than using a loop else clause within a longer function. Next, we looked at the try except else construct. In this case, the else clause is executed only if the try block completed successfully without any exception being raised. This allows the extent of the try block to be narrowed, making it clearer from where we are expecting exceptions to be raised. Python does not have a switch or case construct to implement multi-branch control flow. We showed the alternatives, including chained if elif else blocks and dictionaries of callables. The latter approach also forces you to be more explicit and consistent about what is required and produced by each branch of the control flow, since you must pass arguments and return values rather than mutating local state in each branch. We showed how to implement generic functions, which dispatch on type using the singledispatch decorator available from Python 3. 4. This decorator can be applied only to module scope functions, not methods, but by implementing forwarding methods and argument swapping we can delegate to generic functions from methods. This gives us a way of implementing double dispatch calls. We're done with Python control flow now, and in the next module we'll be looking at low level byte oriented programming. Thanks for watching, and we'll see you in the next module.**

**Byte-oriented Programming**

**Everything Is Bits and Bytes**

**Hello. My name is Robert Smallshire. Welcome to the second module of the Advanced Python course. In this module we'll be going low level to look at byte-oriented programming in Python. At the bottom everything is bits and bytes and sometimes it's necessary to work at this level, particularly when you're dealing with binary or non-text data from other sources. Remember that in Python 3, although not in Python 2, there is a very clear separation between texts, which is stored in the Unicode capable str type and raw bytes, which are stored in the aptly named bytes type. Specifically in this module, we are going to review the bitwise operators, look at the binary representation of integers, fill in some additional details about the bytes type, introduce the bytearray type, look at packing and unpacking binary data with the struct module from the Python standard library, look at memoryview objects, and show how to use memory-mapped files.**

**Bitwise Operations on Integers**

**Let's start right at the bottom with the bitwise operators. These will seem straightforward enough, but our exploration of them will lead us in some murky corners of Python's integer implementation. We covered the bitwise And, bitwise Or, and Shift operators in our Python Fundamentals course when reading binary BMP image files. Now we'll complete the set by introducing the bitwise Exclusive-Or operator and the bitwise compliment operator. Along the way we'll also use bitwise shifts. We'll demonstrate how each of these work, but this will necessitate an interesting detour into Python's integer representation. Recall that we can specify binary literals using the 0b prefix, and we can display integers in binary using the built-in bin function. The bitwise exclusive or operator behaves exactly as you would expect, setting bits in the output value if exactly one of the corresponding operand bits are set. The bitwise compliment or not operator is more unexpectedly difficult to demonstrate, although very easy to use. Here we try to take the compliment of 11110000. You were probably expecting a result of 00001111, although of course, Python doesn't usually print leading 0s, sub 0 b1111 is perhaps more reasonable. Actually, leading 0s are part of the reason we get the surprising -11110001 result. Computer systems typically represent negative integer numbers using a system called Two's Complement. Here's a refresher on how Two's Complement works. Let's see how to represent the number minus 58 in 8 bit Two's Complement, which can represent numbers from -127 to +128. We start with a signed decimal, -58, the value we want to represent. We take the absolute value, which is 58, and represent this in 8 bit binary, which is 00111010. The same binary pattern can be interpreted as the unsigned decimal 58. Now, as the first part of the Two's Complement operation, we flip all the bits, a bitwise not operation to give 11000101. This is where the two in Two's Complement comes from. The complement operation is done in base two, binary. This gives a bit pattern that would be interpreted as 197 as an unsigned integer, although that's not too important here. Finally, we add 1, which gives the bit pattern 11000110, which could be interpreted as 198 is an unsigned integer, which notice, is outside the range -127 to +128 allowed for the signed integers. This is the Two's Complement 8 bit representation for -58. Two's Complement has particular advantages over other representations, such as sign-bit and magnitude schemes. For example, Two's Complement works naturally with arithmetic operations, such as addition and subtraction involving negative numbers. Recall that Python 3 uses arbitrary precision integers, that is, Python integers are not restricted to 1, 2, 4, or 8 bytes, but can use as many bytes as are necessary to store integers of any magnitude. However, Two's Complement presents problems when used with unlimited precision. When we take the complement by flipping the bits, how many leading 0s should we flip? In a fixed precision integer representation the answer is obvious, not so with variable bit width integers, and when you are flipped an infinite number of leading 0s to give an infinite number of leading 1s how do you interpret the results? Of course, Python doesn't really represent negative integers using an infinite number of leading 1s, but conceptually this is what's going on. This is why when asked to represent negative numbers in binary Python actually uses a leading unary minus with magnitude representation, rather than gibing the unsigned binary Two's Complement representation. This means we don't easily get to see the internal bit representation of negative numbers using bin. In fact, we can't even determine what internal representation scheme is used. When our use of the bitwise operators, such as the bitwise not operator, results in a bit pattern that would normally represent a negative integer in Two's Complement format, Python displays that value in sign magnitude format, obscuring the result we wanted. To get the actual bit pattern we need to do a little extra work. Let's return to our earlier example. Let's make v equal to the value 11110000 in binary. This is equal to 240 in signed decimal. Now we'll use the bitwise not operator. Working in binary, we need an 8-bit representation for 240, although to accommodate the Two's Complement representation for the positive integers we need to accommodate at least one leading 0, so really we need at least a 9-bit representation. It is the bits of this 9-bit representation to which the bitwise not is applied, giving 100001111. This is the Two's Complement representation of -241. Displayed back in the signed magnitude representation Python uses when displaying binary numbers, -241 is displayed as -0b11110001. So much for an explanation, but how do we see the flipped bits arising from our application of bitwise not in a more intuitive way? One approach is to manually take the Two's Complement of the magnitude or absolute value of the number. Unfortunately, this also uses the bitwise not operator, and so we end up chasing our tail trying to escape the cleverness of Python's arbitrary precision integers if we try this in Python 3. Another approach is to rely on the characteristics of Two's Complement arithmetic. Take the signed interpretation of the Two's Complement binary value, in this case -241, add to it 2 to the power of the number of bits used in the representation, excluding the leading 1s, so that's 2 to the 8 or 256. Minus 241 plus 256 is 15, and 15 has the 00001111 bit pattern we're looking for, the binary value we expected to get when we applied bitwise not to 11110000. Remember that the Two's Complement of a positive number is itself, so our function needs to take account of this. Also see how we use bitwise shift to raise two to the power of number of bits by shifting the value one to the right by the exponent of eight places. If you have difficulty understanding how this works, don't worry. It is tricky, and 15 minutes with a pencil, paper, and a Python interpreter will be well rewarded. A less usable, but perhaps more obvious approach, is to use bitwise and with a mask of ones to discard all the leading ones in negative results. This effectively specifies to how many bits of precision we want the result to be presented. See how asking for a 9-bit result reveals the leading 1 of the Two's Complement representation? In fact, since Python 3. 2 we can ask Python how many bits are required to represent the integer value using the bit\_length method of the integer type, although note that this excludes the sign. Bit\_length of 32 is 6, the bit\_length of 240 is 8. The bit\_length of -241 is also 8, and the bit\_length of 256 is 9.**

**Byte-wise Operations with Integers**

**We can also retrieve a byte oriented representation directly as a bytes object using the 2 bytes method. Note we are required to pass the length parameter to specify how many bytes we want in the result, and the byte order parameter to specify whether we want the bytes returned in big endian order with the most significant byte first or little endian order with the least significant byte first. If you want to use the native bytes order for your machine you can retrieve the sys. byteorder value. Our machine is little endian. Given some bytes, we can also turn them back into an integer using the complimentary class method from bytes, which accepts a bytes object and a byteorder string, and we can convert the result back into the hex we started with using the hex built-in function. Asking for the byte oriented representation of an integer using to\_bytes will fail for negative integers with an OverflowError; however, if we set the optional signed argument to true, rather than its default value of false, we can get a Two's Complement representation back. This indicates another way to answer the question that started this quest by indexing into the result bytes object to retrieve the least significant byte, and converting that to a binary representation, although concise, this is not.**

**The Bytes Type in Depth**

**Let's take a look at the bytes type in-depth. We first introduced the bytes type way back in the early part of our Foundational Python Fundamentals course. We did this to highlight the essential differences between str, which is an immutable sequence of Unicode code points, and bytes, which is an immutable sequence of, well, bytes. This is particularly important if you're coming to Python 3 from Python 2 where str behaved differently. At this point, you should be comfortable with the bytes literal, which uses the b prefix. The default Python source coding coding is UTF-8, so the characters used in a literal byte string are restricted to printable 7-bit ASCII characters. That is, those with codes from 0 to 127 inclusive, which aren't control codes. This is okay because it's 7-bit ASCII. 7-bit control codes or characters which can't be encoded in 7-bit ASCII result in a syntax error. Norwegian characters like Å and Ø are not 7-bit ASCII. To represent other bytes with values equivalent to ASCII control codes and byte values from 128 to 255 inclusive, we must use escape sequences with \x. Notice that Python echoes these back to us as escape sequences too. This is just a sequence of bytes, not a sequence of characters. If we want a sequence of Unicode code points we must decode the bytes into a text sequence of the str type, and for this we need to know the encoding. In this case, I'll use latin1. When we retrieve an item from the bytes Offline Bundle by indexing we get an int object, not a 1-byte sequence. This is another fundamental difference between bytes and str. Slicing of bytes objects, however, does return a new bytes object. There are a few other forms of the bytes constructor it's good to be aware of. You can create a 0 length byte sequence simply by calling the constructor with no arguments. You can create a 0 field sequence of bytes of a given length by passing a single integer to the bytes constructor. You can also pass an iterable series of integers. It's up to you to ensure that the values are non-negative and less than 256 to prevent a ValueError being raised. One option, if you need to construct a bytes object by encoding a Unicode str object is to use the two argument form of the bytes constructor, which accepts a str in the first argument, and the name of an encoding for the second. Finally, there's a class method, which is a named factory function for creating a bytes object from a string consisting of concatenated, two-digit, hexadecimal numbers. There isn't a method to go in the other direction, so we have to use something like this where we use a generator expression to convert each byte to its hex representation, stripping the leading ox from each resulting string using slicing.**

**The bytearray Type**

**Now let's look at the compliment to the immutable byte sequence, the mutable bytearray sequence. The byte type we just looked at is only one of several so called binary sequence types in Python, whereas bytes is immutable, the bytearray type is mutable. This means it supports many of the same operations as the other mutable sequence type with which you're already deeply familiar, list. The bytearray type supports the same constructors as bytes; Construction of an empty bytearray, construction of a bytearray containing a given number of 0 bytes, construction from another sequence of bytes, such as a bytes object, construction from a Unicode string providing you supply an encoding, and using the fromhex named constructor, a string containing concatenated two-digit hex numbers. As bytearray is mutable, we can, of course, use any of the mutable sequence operations to modify the bytearray in place. Let's create a byte array containing the quick brown fox. We'll extend this sequence using the extend method with jumps over the lazy dog, not forgetting the leading space. As our pangram object is mutable, we can even assign to slices. Let's replace dog with god. The bytearray type effectively supports the same operations as supported by list, together with many of the string like operations supported by bytes, such as upper. We can convert our pangram into upper case. Of course, this uses the notions of upper and lower case, which make sense for ASCII only. We can split our bytearray on white space, which returns a list of bytearray objects, and we can join that list of words back together again, using the join method on bytearray. When using string like operations, such as upper or capitalize on the bytes and bytearray types you must take particular care that only 7-bit ASCII byte strings are in play.**

**Interpreting Binary Structures**

**The Python standard library struct module is so called because it's capable of decoding the byte pattern of struct objects from the C and C++ languages. Structs, short for structures, are composite data types, which consist of the byte patterns for C language primitives concatenated together. It may not be immediately obvious why this is necessary or even useful in a language like Python, but being very close to the machine level, C data types are something of a low level lingua franca for binary data exchange between programs in many languages. Furthermore, as most operating systems are written in C or C++, the ability to interpret C structures into Python values and back again, takes on even greater significance. In this demo we're going to write a binary file from a C program, read the file from a Python program, create Python classes, which mirror C data structures, and demonstrate diagnostic techniques for dealing with binary data. This isn't a course on C, but we're sure you'll understand the following C structures for vector, color, and vertex in this program written in the C99 variant of the language. As you can see, we declare a vector to be comprised of three float values. At this point, it's important to realize that a C float is actually a single precision floating point value represented with 32-bits, which is different from a Python float, which you'll recall is a double precision floating point value represented with 64 bits. We then declare a color structure, which comprises three, unsigned, short integers. Again, this integer type is quite different from what we have available in Python, where we have arbitrary precision signed integers. In fact, C's unsigned short in can only represent values from 0 through to 65, 535 with only 16-bits of precision. The third structure we declare is a vertex, which combines a vector and a color together into one larger structure. In the main function this program creates an array of four vertex structures and writes them to a file called colors. bin before exiting. We'll now compile the C program into an executable. The details of how you do this are heavily system dependent, and require that you at least have access to a C99 compiler. On our Mac OS X system with the Xcode development tools installed we can simply use make from the command line. Make colorpoints compiles colorpoints. c into an executable called simply, colorpoints. When we run the executable a colors. bin file is produced, as expected. Now we have a binary file. Let's try to make sense of it in Python. Our starting point will be a simple program to read only the first vertex from the file. The program will be called reader. py. In the main function we open the file for read, being careful to remember to do so in binary mode. Then we use the read method of file objects to read the entire contents of the file into a bytes object. We use the struct. unpack\_from function to convert the raw byte sequence into more friendly types. This function accepts a special format string containing codes which specify how to interpret the bytes. The leading at character specifies that native byte order and alignment are to be used. Other leading characters can be used to force particular byte orderings, such as less than for little-endian, and greater than for bit-endian. It's also possible to choose between native alignment and no alignment, a topic we'll be revisiting shortly. If no byte order character is specified, then at, the native byte order, is assumed. In our example each of the three f characters tells struct to expect a single precision c float, and each of the upper case H characters tells struct to expect an unsigned short int, which is a 16-bit type. There are also code letters corresponding to all of the common C datatypes, which are mostly variations on different precisions of signed and unsigned integers together with 32 and 64-bit floating point numbers, bytearrays, pointers, and fixed length strings. Running our program, we can see that struct. unpack\_from returns a tuple. The reason our values don't look exactly the same way as they did in the source code to our C program is because we specified the values in decimal in the source, and the values we chose are not representable exactly in binary. There has also been a conversion from the single precision C float to the double precision Python float, which is why the values we get back have so many more digits. Of course, the 16-bit unsigned short int values from C can be represented exactly as Python int objects. One obvious improvement to our program, given that unpack\_from returns a tuple, is to use tuple unpacking to place the values into named variables. Here we use x, y, z, red, green, and blue. We can also shorten our format strings slightly by using repeat counts. For example, 3f means the same as fff. That's no big win, in this case, but it can be very useful for larger data series. Finally, of course, we'd like to read all four vertex structures from our file. We'll also make the example a bit more realistic by reading the data into Python classes, which are equivalents of the vector, color, and vertex structs we had in C. Each class specifies simple data objects with just a dunder in it to store instance variables and a dunder wrapper to display them. We also make a factory function to construct instances of the type vertex, which aggregates a vector and a color. We'll add an import for pretty printing at the top of the module, and then we'll rework the main function to use struct. iter\_unpack, which iterates over binary objects with an identical format specification. When we've accumulated all of the vertices into a list we pretty print the results in data structure. In fact, we can simply unwind one of our earlier refactorings and simply unpack a fields to pull directly into the arguments of make\_colored\_vertex using extended call syntax. In this code fields will be the tuple of six floating int values returned for each structure. Rather than unpacking into named variables we use extended call syntax to unpack the field's tuple directly into the arguments of make\_colored\_vertex. Let's try it. Oh dear. What happened? The struct. iter\_unpack function is complaining because it expects the buffer byte sequence to be a multiple of 18 bytes long. Each of our structures consist of three 4 byte floats and three 2 byte unsigned short integers; (3\*4) + (3\*2) is indeed 18, so how long is our buffer? Let's add some temporary diagnostic code to our program. After we read the file we will print the buffer length and the buffer contents. Curiously, the buffer is reporting as containing 80 bytes, and checking at the command line we can see that that's consistent with the file length. We're expecting 4 lots of 18, which is 72 bytes, so where are the extra 8 bytes coming from? Some investigation is required. It's really awkward to read standard bytes representation, especially when it contains a mix of printable ASCII characters and escape sequences. We can't directly convert a binary sequence into a readable hex string, but Python 3 has some tools in the standard library to help in the form of the bin ASCII module, which contains the oddly named hexlify function. Let's import it with from binascii import hexlify, and modify our diagnostic print statement to print hexlify buffer. This gives us a long hex string, which perhaps isn't even an improvement. What we really want to do is to split pairs of digits with white spaces. We can do this by slicing out consecutive pairs. In this code we hexlify, then decode to an ASCII string. We then join successive two digit slices with spaces using a range expression with a step of two. We then print the hex pairs. This is a big improvement, but it still leaves us counting bytes on the display. Let's precede our line of data with an integer count. Regenerate integers using a range, convert them to strings, and pad each number with leading 0s to a width of 2, good enough for the first 100 bytes of our data. Finally, we have something we can work with. It's slightly annoying that PyCharm interleaves output from the standard out and standard error streams in non-chronological order, but we can live with that for diagnostic code. Now we've got a useful way of viewing our bytes object. Let's get back to diagnosing our problem of why we have 80 bytes rather than 72. Looking carefully, we can see that the first bytes at indices 0 to 17 inclusive contain legitimate data, and we know this to be the case because we decoded it earlier. Looking at bytes 18 and 19, though, we see two 0 bytes. From bytes 20 to 37 inclusive we have another run of what looks like legitimate data, again, followed by another two 0 bytes at indices 38 and 39. This pattern continues to the end of the file. What we're seeing is padding added by the C compiler to align structures on 4 byte boundaries, and our 18 byte structure needs to be padded with 2 bytes to take it to 20 bytes, which is divisible by 4. In order to skip this padding we can use X, which is the format code for pad bytes. In this case, there are two pad bytes per structure, so we can add xx to our format string. With this change in place we can successfully read our structures from C into Python. You might be wondering why, given that we use the @ character at the beginning of our format string to specify native byte order, native size, and native alignment, this didn't work out of the box. So did we. Eventually we traced this mismatch to the fact that our Python interpreter, which is itself implemented in C, and our little C program for writing vertices to a file, were compiled using different C compilers with different structure-padding conventions. This just goes to show that when you're dealing with binary data you need to be very careful if you want your programs to be portable between systems and compilers.**

**The memoryview Type**

**Python has a built-in type called memoryview, which wraps any existing underlying collection of bytes, which supports something called the buffer protocol. The buffer protocol is implemented at sea level inside the Python interpreter and isn't a protocol in the same sense that we use the word when talking about Python level sequence and mapping protocols. In fact, the memoryview type implements the Python level sequence view protocol, allowing us to view the underlying byte buffer as a sequence of Python objects. Our previous example required that we read the data from the file into a byte array, and then translated it with struct unpack into a tuple of numeric objects, effectively duplicating data. We're going to change that example now to use memoryviews, avoiding the duplication. We can construct memoryview instances by passing any object that supports the buffer protocol C API to the constructor. The only built-in types which support the buffer protocol are bytes and byte array. We'll construct a memory view from the buffer just after our diagnostic print out with this line of code; mem = memoryview(buffer). To explore the capabilities of memoryview we could use a debugger, such as PDB or the one here in PyCharm to stop the program just after this line, but we'll introduce you to another technique from the Python standard library code module, code. interact. This function will suspend the program and drop us to the REPL. By passing a reference to the local namespace we can get access to the live variables in our program, including our memoryview object, mem. We use a call to the local's built-in function to get a reference to the current namespace. Now when we run our program we get a REPL prompt at which we can access our new mem object. The memoryview object supports indexing, so retrieving a byte at index 21 and converting to hexadecimal gives us 3f, which is what we would expect. Being bona-fide sequences memory view objects support slicing. Here we slice from bytes 12 to 18. Crucially, memory view slices are also memory views. There's no copying going on here. The 0th byte of the slice has value e0. Much in the same way as we use the struct module to interpret binary data as native types, we can use the memory view cast method to do the same. This method accepts the same format codes as are used by the structs module, expect that only a single element code is permitted. In other words, the interpreted items must all be of the same type. We know that the bytes in the 12 to 18 slice represent 3 unsigned short int values, so by passing capital H to the cast method we can interpret the values that way. Notice that cast also returns a memoryview, but this time one that knows the type of its elements. This gives us the same unsigned integer values that we wrote from our original C program that created the binary file. Alternatively, we can use the memory view to list method to convert the series of elements to a list. Now we're finished with the interactive session provided by Code Interact. We can send the end of file character, just as you normally would to terminate a REPL session, Ctrl+D on Unix or Ctrl+Z on windows. Your program will then continue executing. Before moving on we'll remove the code interact line, so our program runs to completion and uninterrupted next time. Let's use the ability to slice and cast memory views to modify our vector and color types to use the bytes buffer as the underlying storage. Our modified vector class now looks like this. The initializer accepts a memoryview, which we expect to expose floats. We validate this by checking the format code returned by the memoryview format attribute against a string containing f and d. Those code for single and double precision floats respectively. We also checked that the memoryview exposes at least three items. Note that when used with memoryview len returns the number of items, not the number of bytes. The only instance attribute of vector\_mem now holds a reference to the memoryview. Our old instance attributes are replaced by read only properties which perform the appropriate lookups in the memoryview. Since we can use properties to replace attributes our dunder repr implementation can remain unmodified. The modified color class works exactly the same way, except now we check that the memoryview wraps unsigned integer types. Our vertex class, which simply combines a vector and a color can remain unchanged, although our make\_colored\_vertex factory function needs to be changed to accept a memory view, specifically one that is aligned with the beginning of a vertex structure. The function now slices the vertex memory view into two parts for the vector and color respectively, and casts each to a typed memory view. These are used to construct the vector and color objects, which are then passed on to the vertex constructor. Back in our main function, after creation of the mem instance, we'll need to rework our main loop. We start by declaring a couple of constants describing the size of a vertex structure, and the stride between successive vertex structures. This allows us to take into account the 2 byte padding between structures. Next, we'll set up a generator expression which yields successive memory views into whole vertex structures. Remember that each slice into mem is itself a memoryview. This time rather than an expressive for loop to build the list of vertices we'll use a list comprehension to pass each vertex memoryview in turn to make colored vertex. Running this program, we can see we get exactly the same results as before, except that now our vector and color objects are backed by the binary data we loaded from the file with much reduced copying.**

**Memory-mapped Files**

**There's still one copy happening though, the transfer of bytes from the file into our buffer bytes object. Not a problem for our trifling 80 bytes, but for very large files this could be prohibitive. By using an operating system feature called memory mapped files we can use the virtual memory system to make large files appear as if they are in memory. Behind the scenes the OS will load, discard, and sync pages of data from the file. The details of this are operating system dependent, but the pages are typically only 4Kb in size, so this can be memory efficient if you need to access relatively small parts of large files. Memory-mapped file support in Python is provided by the standard library mmap module, which contains a single class, also called mmap, which behaves like a mix between a byte array and a file like object. The mmap instance is created around a file handle on Windows or a file descriptor on Unix, either of which can be obtained by calling the fileno method of a regular file object. In fact, mmap instances support the C API buffer protocol, and so can be used as the argument we pass to our memoryview constructor. Let's modify our example program to do so. All this requires is that we import the mmap module at the top of our file and modify our main function to retrieve the file handle or descriptor, passing it to the mmap constructor. As with any other file-like object, mmap objects must be closed when we're done with them. We can either call the close method explicitly or, more conveniently, use the mmap object with a context manager. We'll do the latter. The only difference here is that buffer is now a memory map file rather than the byte sequence, as it was previously. We've avoided reading the file into memory twice; once into the OS maintained file cache, and again, into our own collection by directly working on the operating systems view of the file. This works after a fashion when we run it, insofar as our vertex objects are created with the memory-mapped file backing store; however, we get a nasty failure when the mmap object is closed by the context manager, which tells us that exported pointers exist. The reason for this is that at the point the mmap object is closed by the context manager we still have a chain of extend memoryview objects which depend ultimately on the mmap object. A reference counting mechanism in the buffer protocol has tracked this and knows that the mmap still has memoryview instances pointing at it. Thinking about our live object graph, there are two local variables, which ultimately hold references to the memory map file, mem, which is our lowest level memoryview, and vertices, which is the list of vertex objects. There are a couple of approaches we could take here. We could arrange for the memoryview release method to be called on the memoryview object inside our vector and color instances. This method deregisters the memoryview from any underlying buffers and invalidates the memoryview, so any further operations with it raise value errors. This would just move the problem though. Now we'd have zombie vector and color instances containing invalid memoryviews. By explicitly removing these name bindings, using two invocations of the del statement, we can clean up the memoryviews, so the memory map can be torn down safely. Better to respect the constraint in our design, the lifetime of our memory-mapped file backed objects must be shorter than the lifetime of our memory mapping. With this change in place our program runs flawlessly.**

**Summary**

**Let's summarize what we've covered in this module. Our review of Python's bitwise operators, including exclusive or and compliment, led us down the path of understanding how Python internally represents and externally presents arbitrary precision integer values at the bit level. To print bit patterns for values which equate to negative numbers you can sign extend the results by using bitwise and with a binary value consisting of 1 bits. We reviewed the constructors and some key methods of the immutable binary sequence type bytes and the mutable binary sequence type, bytearray. We showed how to interpret a series of bytes as native or C language types, such as short unsigned integer, translating them into compatible Python types using the struct module of the Python standard library. We demonstrated how to display byte streams in an easily readable hexadecimal format using the hexlify function in conjunction with some simple formatting expressions. We explain that a detailed understanding of how C compilers align data structures on certain byte or word boundaries can be crucial to correctly reading the data files produced by C or C++ programs. We introduced memoryview for obtaining copy free views, slices, and type casts from underlying byte sequences. We touched on the use of the really useful interact function from the standard library code module, which allows us to drop to a REPL at any point in the program and resume execution later, and we showed that memory-mapped files implemented with mmap module implement the buffer protocol, and so can be used as the basis for memory view objects, further reducing the amount of data copying when reading files. There's a lot to be learned about low-level, byte-oriented programming in Python, including writable memory views, shared memory with mmap, and interfacing to native C and C++ code. We can't possible cover all that in this course, but we've shown you the starting points for your own explorations. Thanks for watching, and we'll see you in the next module.**

**Object Internals and Custom Attributes**

**Introducing Object Internals**

**Hello. My name is Robert Smallshire. Welcome to the third module of the Advanced Python course. In this module we'll provide details of how objects are represented internally in Python as a dictionary called dunder dict, and show how objects can be directly queried and manipulated through this internal dictionary. We'll then use this knowledge to show how to implement custom attribute access by overriding the special dunder getattr, dunder getattribute, dunder setattr, and dunder delattr methods. We'll round off by seeing how to make objects more memory efficient by defining dunder slots.**

**How are Python Objects Represented?**

**Let's start with a simple class to represent two-dimensional vectors called simply, vector. This has an initializer, which accepts the two components of the vector, x and y, and these are stored on two instance attributes also called x and y. We also include a dunder wrapper, so that we can print a nice textural representation of our vector objects. Let's instantiate an object of this class in the REPL with components 5 and 3. In the list returned by the built-in dir function we see two named attributes, x and y, along with many of Python's special dunder attributes, quite a few of which we've explained previously in Python Fundamentals of Python Beyond the Basics. One attribute in particular is of interest to us today, and that is dunder dict. Let's see what this is. As its name indicates, dunder dict is indeed a dictionary, one which contains the names of our objects attributes as keys, and the values of our objects attributes as, well, values. Here's further proof, if any were needed, that dunder dict is a Python dictionary. We can retrieve the type of v. dunder dict, which is dict. We can retrieve attribute values directly from dunder dict by indexing into the dictionary with square brackets, just like a normal dictionary because it is a normal dictionary. We can even modify attribute values through dunder dict. Going further, we can even remove attributes entirely. As you might by now suspect, we can also test for their existence using the in operator, and add new attributes into the dictionary, and thereby into our object. Although all of these direct queries and manipulations of dunder dict are possible, for the most part, you should prefer to use the built-in functions getattr, hasattr, delattr, and setattr. Direct access to dunder dict does have legitimate uses though, and it's essential to be aware of it's existence and how and when to use it for advanced Python programming. Our vector class, like most vector classes, has hard wired attributes called x and y to store the two components of the vector. Many problems, though, require us to deal with vectors in different coordinate systems within the same code or perhaps it's just convenient to use a different labeling scheme, such as u and v instead of x and y in a particular context. Let's see what we can come up with. In this code we accept a double star keywords args argument, which is received into the coords dictionary, which we will then use to update the entries in dunder dict. Remember that dictionaries are unordered, so there's no way to ensure that the coordinates are stored in the order they are specified as actual function arguments. Our dunder repr implementation must iterate over the dictionary, sorting by key for convenience. Our modified vector class allows us to provide arbitrary coordinate names. Here we set p to three and q to seven. This is all very well, but our coordinates are now essentially public attributes of our vector objects. What if we want our vector class to be an immutable value type, so we provide values to the constructor, which can't be subsequently changed? Ordinarily we will do this by prefixing our attributes with an underscore to signal that they are implementation details, and then provide a property with only a getter to prevent modification. In this case, though, we don't know the attribute names in advance, so we can't declare a property getter, which must be named at class definition time, not at object instantiation time. We'll show how to work around this using the special dunder getattr method, but first let's change our dunder init method to store data in private attributes and add dunder repr to report them correctly. In this version of the code we use a dictionary comprehension to associate new names with a prefix underscore with each existing value and assign this to a new dictionary called private\_coords. This is what we then use to update dunder dict. In the dunder repr code we must remember to strip the leading underscore when displaying the representation, in this case, by using string slicing to emit the first character. We can construct and represent vector instances just as before. Here we use p=9 and q=3, but now the attributes are stored in private attributes called \_p and \_q. Because of this we can no longer access Python directly. It doesn't exist as an attribute.**

**Overriding \_\_getattr\_\_**

**What we'd like to do is to fake the existence of p for read access, and to do that we need to intercept attribute access before the AttributeError is raised. To do that we could use dunder getattr. Notice that there are two very similarly named methods, dunder getattr, and dunder getattribute. The former is only called when regular attribute lookup fails. The latter is called for all attribute access, irrespective of whether an attribute of the requests name exists of not. For our case, we'll be using dunder getattr to intercept lookup failures. Here's our vector definition, which incorporates dunder getattr. We've just added a simple stub that prints the attribute name. When we request non-existent attributes the name of the attribute is now printed. Here we ask for p and it prints name = p, and when we ask for q it prints name = q, but when we request the name of an attribute that does exist we simply get the attribute value, indicating that dunder getattr isn't being called in these cases. Now we can modify dunder getattr to prepend the underscore to the name, and retrieve the underlying attribute for us. At first sight, this appears to work just fine. Now we can retrieve p and q, but there's some serious problems lurking here. The first is that we can still assign to p and q. Here we set p to 13. Remember that there wasn't really an attribute called p, but Python has no qualms about creating it for us on demand. Worse, because we have unwittingly brought p into existence, dunder getattr is no longer invoked for requests for p, even though our hidden attribute, \_p, is still there behind the scenes with a different value. Currently, p has a value of 13, and \_p has a value of 5.**

**Overriding \_\_setattr\_\_**

**We can prevent this by overriding dunder setattr too to intercept attempts to write to our attributes. In this case, dunder setattr is configured to simply raise an AttributeError with a message explaining that we're not allowed to set attributes on immutable objects of this class. This successfully prevents writing to any attribute. We'll return to dunder setattr shortly, but first we need to demonstrate another problem with our dunder getattr.**

**Pitfalls with \_\_getattr\_\_**

**Although dunder getattr works fine with attributes we've carefully faked, such as p, look what happens when we try to access an attribute for which there is no fake support, such as x. We get a runtime error, maximum recursion depth exceeded while calling a Python object. In other words, we've got infinite recursion. This happens because our request for attribute x causes dunder getattr to look for an attribute, \_x, which doesn't exist. This in turn invokes dunder getattr again to look up attribute \_\_x, which doesn't exist, and so on recursively until the Python interpreter exceeds its maximum recursion depth and raises a runtime error. To prevent this happening you might be tempted to check for the existence of the private attribute using hasattr, like this, raising an AttributeError if the private attribute can't be found. Unfortunately, this doesn't work either, since it turns out that hasattr also ultimately calls dunder getattr in search of the attribute. What we need to do is directly check for the presence of our attribute in dunder dict using the not in operator. This now works as we would wish. We can create a vector with components p=9 and q=14. We can display the vector, which echoes those values back to us. We can query individual components, such as p, which is of course equal to nine, but when we request x, which is not defined, an AttributeError is raised, as we would expect. In fact, attribute lookup in Python follows a pretty complex procedure, so instead of invoking that procedure again, by calling getattr, we can just directly return the attribute from dunder dict. This also enables us to switch to easier to ask forgiveness than permission, EAFP style programming, rather than look before you leap, LBYL style programming. Now the code attempts to retrieve the attribute with a private name directly from dunder dict. If it's not there a KeyError is raised, and in the except KeyError block we convert that to an AttributeError. Of course, the easier to ask forgiveness than permission version has exactly the same behavior as the look before you leap version.**

**Overriding \_\_delattr\_\_**

**Now let's look at customizing attribute deletion. Deleting attributes is something that is very rarely seen in Python, although it is possible. Here, using a vector, v, with components p=1 and q=2, our vector class also allows us to remove attributes by calling delattr. We rather mischievously delete the \_p attribute using a call to delattr and the \_q attribute by using the built-in del statement. This leaves us with a vector with no components. Although a client is unlikely to attempt this inadvertently, we can prevent it by overriding dunder delattr. In a similar manner to our dunder setattr, our dunder delattr simply raises an AttributeError with a message informing the client that the operation is forbidden. Here's how that code behaves when trying to delete either public or private attributes using a vector, v, with components p=9 and q=12. Deleting q using the del statement with v. q causes an AttributeError to be raised.**

**Customizing Attribute Storage**

**There is no requirement for us to store attributes directly in dunder dict. Here's an example of a subclass of vector called ColoredVector that stores immutable red, green, blue color along with the immutable vector components. Internally we store the red, green, blue channels in a list, which is assigned to the attribute called color. We override both dunder getattr and dunder setattr to provide read and write access to the color channels, being careful to forward requests to the super class when necessary. Here we create a ColoredVector called cv with color components red, 23, green, 44, blue, 238, and special components p=9, and q=14. Our code allows us to retrieve these attributes, red, green, p, and q as if they were just regular attributes, although the storage mechanisms for the vector components and the color components are quite different. When we do dir(cv) we see \_p and \_q for the vector components, and color for the color components. There's a gremlin lurking here though. Our dunder repr implementation in the base class makes an assumption, which is no longer valid. It assumes all attributes are prefixed with an underscore internally, and doesn't know about color. This causes our dunder repr to behave in unexpected ways. This is surprisingly hard to fit elegantly without the derived ColoredVector class knowing too much about implementation details of the vector base class. We believe it should be possible to derive from a class without knowing how it works. The base class dunder repr makes assumptions about the contents of its dunder dict, which it cannot reasonably expect to be respected by subclasses. As an exercise, we recommend changing vector, so it stores it's components in a dedicated dictionary separate from dunder dict, although of course, this dictionary itself will need to be stored in dunder dict. Here's another fix which works by duplicating and modifying some of the logic in the base class into an override of dunder repr. This method override works by removing the attribute named color from the list of keys before using the same logic as the super class to produce the string of sorted coordinates. The color channel values are accessed in the normal way, which will invoke dunder getattr. It's worth bearing in mind that this example demonstrates the awkwardness of inheriting from classes, which were not deliberately designed as base classes. Our code serves its purpose in demonstrating customized attribute access, but we couldn't recommend such a use of inheritance in production code.**

**Direct vs. Indirect Access to \_\_dict\_\_**

**Do you remember the vars built-in function? Without an argument it returns a dictionary containing the current namespace, and so acts just like the locals built-in function, however, if we supply an argument it returns the dunder dict attribute of its argument, so instead of writing obj. dunder dict we can write vars(obj). Arguably, this is more Pythonic than accessing dunder dict directly, for much the same reason that calling len(collection) is definitely more idiomatic than calling collection. dunder len. That said, the length returned by len is always immutable, whereas dunder dict is a mutable dictionary. In our opinion, it's much clearer that the internal stage of an object is being modified when we directly modify the dunder dict attribute like this by assigning to one of its keys, then when we go via vars. Whichever form you use, and we don't feel strongly either way, you should be aware of this use of vars with an argument.**

**Overriding \_\_getattribute\_\_**

**Now let's look at dunder getattribute as opposed to dunder getattr. Recall that dunder getattr is called only in cases when normal attribute lookup fails. It's our last chance for us to intervene before the Python runtime raises an attribute error, but what if we want to intercept all attribute access? In this case, we can override dunder getattribute. I use the term override advisedly because it's the implementation of dunder getattribute in the ultimate base class object that is responsible for normal lookup behavior, including calling dunder getattr. This level of control is seldom required, and you should always consider whether dunder getattr is sufficient for your needs. That said, dunder getattribute does have its uses. Consider this class, which implements a logging proxy, which logs every attribute retrieval made against the target objects supplied to the constructor. Since dunder getattribute intercepts all attribute access through the dot operator we must be very careful never to access attributes of the logging proxy through the dot. In the initializer we use the dunder setattr implementation inherited from the object super class to do our work. Inside dunder getattribute itself we retrieve the attribute called target using a call to the super class implementation of dunder getattribute, which remember, implements the default lookup behavior. Once we have a reference to the target class we delegate to the getattr built-in function. If attribute lookup on the target fails we then raise an attribute error with an informative message. Note how careful we must be even to return our object's dunder class. If attribute retrieval was successful we report as much before returning the attribute value. Let's see this in action. Note we're doing this in the regular Python REPL in a terminal rather than using the PyCharm REPL simply because PyCharm itself performs a great many invocations of dunder getattribute in its console to support code completion. We create a ColoredVector called CV with colors red 23, green 44, blue 238, and vector components p=9 and q=14. We then wrap a LoggingProxy called CW around the ColoredVector, CV. When we retrieve the p attribute from CloudWatch our dunder getattribute code is invoked, and the message is printed. Likewise, for the red attribute. So far so good, but what happens when we write to an attribute through the proxy? In this example both writes appear to be accepted without error, although only one of them should be. Remember, the vector component should be immutable. Only the colors are mutable. In fact, though, neither of the writes succeeded. Let's check by retrieving the attributes. It looked like p was set to 19, but it still has the value 9. The color red should have been set to 5, but it still has the value 23. What's happening here is that our attribute writes to the CW proxy are invoking the inherited dunder setattr on the object base class, which is actually creating new attributes in the logging proxy instances, dunder dict, however, reads through the proxy, correctly bypass this dunder dict, and are redirected to the target. In effect, the proxy dunder dict has become write only. The solution to this is to override dunder setattr on the loggingproxy too. In the implementation we retrieve the attribute called target using a call to the superclass implementation of dunder getattribute. We'll then use the setattr built-in function to set the attribute on the target. With dunder setattr in place we can successfully write through the logging proxy to modify mutable attributes. Here we update red to 55, and attempted writes to immutable attributes, such as p, are rejected as we intended.**

**Attribute Lookup for Special Methods**

**Some special methods appear to bypass dunder getattribute. Let's look at what's going on. It's important to realize that dunder getattribute only intercepts attribute lookup through the dot operator. Let's create a colored vector called cv and a LoggingProxy for it called cw. If we call the dunder repr method directly on our proxy, cw, the call is routed via the proxy, and is dispatched successfully on the target, returning the repr for ColoredVector. However, if we request the repr of cw in the conventional manner using the built-in function the dunder getattribute function of our logging proxy is not invoked. The call is not forwarded to the ColoredVector, and instead we get the default repr for the LoggingProxy object. This demonstrates that dunder getattribute can only be used to intercept special method calls when the special method is retrieved directly, which is something we don't normally do. Normal access to facilities provided by special methods is through the built-in functions, such as len, iter, and repr, and so on. These all bypass the dunder getattribute mechanism for performance reasons. What this means in practice is that if you want to write a proxy object, such as LoggingProxy, which transparently proxies an object, including its repr or other special methods, it's up to you to provide an implementation of dunder repr that forwards the call appropriately. Here is such a forwarding version of dunder repr. It works by using dunder getattribute on the object base class to retrieve the target object, and then using getattr on the target object to retrieve the dunder repr method, which we then call. This now works when called by the repr function, although our proxy has become fairly invisible at this point, which may hinder debugging.**

**Where are Methods Stored?**

**One question you may have is, where are the methods? Why is it that when we introspect the dunder dict of an object we only see attributes and not methods, but as we have just seen with dunder repr, we can retrieve methods using getattr, so where are the methods? The answer is that the methods are attributes of another object, the class object associated with our instance. As we already know, we can get to the class object via the dunder class attribute, and show enough it has a dunder dict attribute, which contains references to the callable objects, which are the manifestations of the methods of our class. Returning briefly to our vector example, we create a vector, v, with Vector components x=3 and y=7. When we retrieve its dunder dict we see \_y and \_x are private attributes. When we retrieve v. dunder class we see the type of the vector object, Vector. We can in turn retrieve the dunder dict from dunder class. This is an object of type mappingproxy. Of course, we can retrieve the callable object and pass that instance to it taking the place of what is normally the self-argument to a method. It's well worth spending some time experimenting on your own and poking around with these special attributes to get a good sense of how the Python object model hangs together. It's worth noting that the dunder dict attribute of a class object is not a regular dict, but is instead of type mappingproxy, a special mapping type used internally in Python, which does not support item assignment. To add an attribute to a class you must use the setattr function. The machinery of setattr knows how to insert attributes into the class dictionary.**

**Trading Size for Dynamism with Slots**

**We'll finish off this part of the course with a brief look at a mechanism in Python for reducing memory use, slots. As we have seen, each and every object stores it's attributes in dictionary. Even an empty Python dictionary is quite a hefty object weighing in at 288 bytes. Here we create an empty dictionary and use the getsizeof function from the sys module to measure its size. If you have thousands or millions of objects this quickly adds up, causing your programs to need megabytes or gigabytes of memory. Given contemporary computer architectures, this tends to lead to reduced performance, as CPU caches can hold relatively few objects. Techniques to solve the higher memory usage of Python programs can get pretty involved, such as implementing Python objects in a lower level language, such as C or C++, but fortunately Python provides the slots mechanism, which can provide some big wins for low effort with tradeoffs that will be acceptable in most cases. Let's take a look. Consider the following class to describe the type of electronic component called a resistor. In our simple model resistors have a resistance in ohms, a tolerance in percent, and a power capability in watts. We simply store these as three instance attributes of the same name. It's difficult to determine the size in memory of Python objects, but with care we can use the getsizeof function in the sys module. To get the size of an instance of resistor we need to account for the size of the resistor object itself and the size of its dict. Here we create a 10 ohm resistor, r10, with a resistance of 10 ohms, a tolerance of 5%, and a power of 0. 25 of a watt. Its size weighs in at 152 bytes. Python objects being highly dynamic, dictionary-based objects we can add attributes to them at runtime and see if this is reflected as an increased size. Let's add a cost attribute to r10 of $0. 02. Now its size has shot up to 248 bytes. This is quite a big object, especially when you consider that the equivalent struct in the C programming language would weigh in at no more than 64 bytes with very generous precision on the number types. Let's see if we can improve on this using slots. To use slots we must declare a class attribute called dunder slots. To dunder slots we assign a list containing the strings, resistance\_ohms, tolerance\_percent, and power\_watts. Now look at the space performance of this new class. We can instantiate resistor, just as before, and retrieve its attributes in exactly the same way; however, its size is much reduced from 152 bytes down to 64 bytes, less than half the size. There's always a tradeoff though, as we can no longer dynamically add attributes to instances of resistor. This is because the internal structure of resistor no longer contains a dunder dict. For most applications slots won't be required, and you shouldn't use them unless measurements indicate that they may help, as slots can interact with other Python features and diagnostic tools in surprising ways. In an ideal world slots wouldn't be necessary, and in our view they're quite an ugly language feature, but at the same time, we've worked on applications where the simple addition of a dunder slots attribute has made the difference between the pleasure of programming in Python and the pain of programming in a lower level, but more efficient language. Use slots wisely.**

**Summary**

**Let's summarize what we've covered in this course module. We've discovered that Python objects store their attributes internally within a dictionary called dunder dict, which maps attribute names to attribute values. We showed that instance attributes can be created, retrieved, updated, and deleted by direct manipulation of dunder dict. We showed how any failure to retrieve an attribute by normal means causes the dunder getattr special method to be invoked. The implementation of dunder getattr can use arbitrary logic to fake the existence of attributes programmatically. Similarly, assignment attributes can be customized by overriding dunder setattr, and deletion of attributes can be customized by overriding dunder delattr. Calls to the hasattr built-in function may also invoke dunder getattr, so dunder getattr implementations need to be particularly careful to avoid non-terminating recursion. Occasionally, it's necessary to customize all attribute lookup, even for regular attributes. In these cases, the default lookup machinery and the special dunder getattribute method of the object based class may be overridden, taking care to delegate to the base class implementation as necessary via a call to super. Method callables are stored in the dunder class. dunder dict dictionary. Slots are a quick way to make Python objects more memory efficient at the cost of them being less dynamic. In the next module we'll be looking at even more sophisticated ways to customize attribute access using a Python feature called descriptors. Thanks for watching, and we'll see you in the next module.**

**Descriptors**

**Introducing Descriptors**

**Hello. My name is Robert Smallshire. Welcome to the fourth module of the Advanced Python course. In this module we'll investigate a feature of Python you've been using, perhaps unknowingly, called descriptors. Descriptors are the mechanism used to implement properties in Python. We covered properties thoroughly in Python Beyond the Basics, but we'll start here with a short recap. In Python Beyond the Basics we showed how to create properties using the property decorator. In this module we'll dig deeper and show how to create properties in the raw using the property constructor. Then we'll show how to create a specialized property by defining a custom descriptor, which implements the descriptor protocol. We'll round off by demonstrating that there are two categories of descriptor, data descriptors and non-data descriptors, and we'll show how these interact with Python's somewhat complicated attribute lookup rules. As promised, we'll start with a very brief review of properties, our entry point into the world of descriptors. To explain descriptors we'll be building this simple class to model planets, such as Pluto here, focusing on particular physical attributes, such as size, mass, and temperature. Let's start with this basic class definition for a planet in planet. py consisting of little more than an initializer. There are no properties here yet. We'll add them in a moment. Inside the planet initializer we initialize the planet's name, its radius in meters, its mass in kilograms, its orbital period in seconds, and its surface temperature in Kelvin. This Planet class is simple enough to use. Here we create an instance of planet called Pluto, which we give a radius of 1184Km, together with the correct values for its other physical quantities. Of course, we can easily retrieve the value of the radius\_metres attribute. Unfortunately, our code allows us to represent nonsensical situations, such as setting a negative radius, either by directly mutating an attribute like this, here we set the radius of Pluto to -10, 000 meters or simply by passing nonsense, such as a 0 mass and negative orbital period or a temperature below absolute 0 to the constructor. We already know how to improve this sort of code, by wrapping our instance attribute in property getters and setters, which perform validation by checking the physical quantities are positive. We then assigned through those properties in the initializer to get validation on constriction for free. Here we've wrapped radius\_metres, mass\_kilograms, orbital or period seconds, and surface temperature kelvin in properties setters and getters. From a robustness standpoint this code is much better. For example, we can no longer construct massless planets. Now we get a value error with a helpful message telling us that a mass in kilograms with a value of 0 is not positive. The tradeoff, though, is that the amount of code is exploded, and worse, there's a lot of duplicated code checking that all those numeric attribute values are non-negative. Descriptors will ultimately provide a way out of this, but first we need to do a little more unraveling of properties to aid our understanding.**

**Properties are Descriptors**

**Back in Python Beyond the Basics we introduced property as a function decorator for property getters. To briefly recap, a getter method, which encapsulates direct attribute access, is decorated by the property decorator, which creates a property object. The getter function is bound to an attribute of the property object called fget, and the original name of the getter is conceptually rebound to the property object. The property object also has an attribute called setter, which is in fact another decorator. When a setter method is decorated by the setter the original property object is modified to bind and attribute called fset to the setter method. The property object effectively aggregates the getter and setter into a single property object which behaves like an attribute because it is a descriptor. Shortly we'll learn how it is able to appear so attribute like, but first let's unravel properties a bit more. Remember that function decorators are just regular functions which process an existing function and return a new object, usually a new function, which wraps the decorated function. Instead of using decorator syntax to apply the decorator to a function we can just define a regular undecorated function, and then pass the function to the decorator, rebinding it to the same or indeed any other name. Given that decorators are just functions, let's rework our code to apply property explicitly using regular function call syntax, avoiding the special decorator application syntax using the @ symbol. When doing this it's important to note that the property function supports several arguments for simultaneously supplying the getter, setter, and deleter functions, along with a doc string value. In fact, help property makes this quite clear. As you can see, we can separately define our getter and setter functions, then call the property constructor within the class definition to produce a class attribute. Let's use this form in our Planet class for the numerical attributes. First we'll remove the property decorator from the getter functions before removing the setter decorator from the setter functions. Now we'll prefix the names of the getter functions with \_get\_ and the names of the setter functions with \_set\_. We use single underscore prefixes here because these are not special methods. Finally, we create a property object using the property function. You can think of it as a constructor call in this context, passing the pair of getter and setter functions. Continuing, we need to do the same for the mass\_kilograms property, the orbital\_seconds property, and the surface\_temperature\_kelvin property. If you think this form of property set up is a retrograde step compared to the decorator form we'd agree with you. There's a good reason we introduce properties as decorators first, nevertheless, it reinforces the notion that property is simply a function, which returns an object called a descriptor bound to a class attribute. The runtime behavior of this code hasn't changed at all. We can still create objects, retrieve attribute values through properties, and attempt to set attribute values through properties with rejection of nonsensical values. We already know what sort of operations we can perform with a descriptor. We can get a value, set a value, and delete a value. In the case of property, these getting, setting, and deleting operations call functions of our own, which query and manipulate instance attributes, although in general, the descriptor operations can be implemented to do almost anything.**

**Implementing a Descriptor**

**We've seen that property is a descriptor, which wraps three functions. Let's create a more specialized descriptor useful for modeling the strictly positive numeric values in our planet class. Here's a simple descriptor called Positive. The descriptor class implements three methods, which comprise the descriptor protocol; dunder get, dunder set, and dunder delete, which are called when we get a value from a descriptor, set a value through a descriptor, or delete a value through a descriptor respectively. In addition, the positive class implements dunder init to configure new instances of the descriptor. Before we look in more detail at each of these methods let's make use of our new descriptor to refactor our Planet class. We remove the setters and getters for radius\_metres and replace the call to the Property constructor with a call to the Positive constructor. We do the same for mass\_kilograms, orbital\_period\_seconds, and surface\_temperature\_kelvin. With the positive descriptor on hand the Planet class shrinks by a huge amount. At first sight, this may appear confusing. It looks like we're assigning to radius\_metres twice, once in the initializer, and once in the body of the class. In fact, the call in the body of the class is binding an instance of a positive descriptor to a class attribute of the Planet. The call in dunder init is then apparently assigning to an instance attribute, although, as we'll see in a moment, this assignment is actually invoking a method on the descriptor object. Allow us to explain the machinery we have created. Let's start with an instance of Planet. When we retrieve an attribute like this, with pluto. mass\_kilograms, the Positive. dunder get function is called. The instance argument is set to Pluto, and the owner argument is set to the object which owns the descriptor. In this case, the class Planet. Similarly, our assignments to the descriptors in the Planet. dunder init function resolved to calls effectively equivalent to Positive. dunder set. Let's review graphically what we have here. We have a reference, Pluto, which is bound to an instance of Planet. The dunder class reference of the instance points to the Planet class object. The Planet class object contains four attributes, each of which references a distinct instance of the Positive descriptor object. The dunder class reference of each descriptor object refers to the Positive class. Notice that because the descriptor is owned by the Planet class rather than by the Pluto instance, we can't just store the value in the descriptor object. If we did that the value would be shared between all planet instances. Instead, we need to somehow associate the attribute value with the instance. At first, this seems easy. The dunder get call is handed a reference to the instance, Pluto in this case, so why not just store the value in Pluto's dunder dict? There's a problem though. Within the descriptor class, Positive, we have no way of knowing to which attribute name the descriptor is bound in the Planet class. We can distinguish between descriptor instances in dunder get using the self-argument, but in Python objects do not know which names have been bound to them. This means we have no way of correlating between descriptor instances and the descriptor names embedded in the Planet class. In other words, none of our descriptor objects know which quantity they represent. This, in turn, means we can't store the descriptor value in the dunder dict of Pluto because we wouldn't know what dictionary key to use. This apparent shortcoming of descriptors not knowing the name of the class attribute to which they are bound is also evident in the fact that our value validation error message in dunder set no longer mentions the attribute name, a clear regression in capabilities from what we had earlier. This is fixable, but the solution will have to wait until the next module in this course when we look at meta classes. So how to associate values with instances? Let's look at the solution pictorially first, then we'll reiterate with code. We use a special collection type from the Python standard library called WeakKeyDictionary. This works pretty much like a regular dictionary, except that it won't retain value objects, which are referred to only by the dictionary key references. We say the references are weak. A weak key dictionary owned by each descriptor instance is used to associate Planet instances with the values of the quantity represented by that descriptor, although the descriptor itself doesn't know which quantity is being represented. For example, the WeakKeyDictionary shown here associates Planet instances with mass\_in\_kilograms values. A separate WeakKeyDictionary associates Planet instances with radius\_in\_metres values. Because the dictionary keys are weak references, if our Planet instance is destroyed, let's pretend the earth is vaporized to make way for a hyperspace bypass, the corresponding entries in all the WeakKeyDictionaries are also removed. Now let's switch over to code to see how it's implemented in practice. A WeakKeyDictionary instance called \_instance\_data is created in the descriptor initializer, so as we've just seen, our Planet class indirectly aggregates four such dictionaries, one in each of the four descriptors. Within the dunder set method we associate the attribute value with the Planet instance by inserting a mapping from the instance as key to the attribute as value. As such, a single dictionary will contain all the radius\_metres values for all planet instances. Another dictionary will contain all mass\_kilograms values for all Planet instances, and so on. We're storing the instance attribute values completely outside the instances, but in such a way as we can reliably retrieve them in dunder get. Just to convince you that all of this works, we'll show it in action in the debugger. We'll stop the program in the initializer for Mars, and step into the descriptor dunder set function for technology radius\_metres attribute. We can see that the \_instance\_data dictionary already contains the three radii for Mercury, Venus, and Earth.**

**Calling Descriptors on Classes**

**One aspect of the descriptor protocol we haven't yet addressed is what happens when we retrieve a descriptor from a class. As we've seen, instance attribute retrieval works fine; however, class attribute retrieval does not. In such cases, the instance argument of dunder get will be set to none, which causes a failure with the WeakKeyDictionary used for attribute storage because we cannot create a weak reference to none. We can use this fact to detect when a descriptor value is being retrieved via a class attribute by testing the instance argument against none. In the dunder get implementation, when instance is none we do what seems most natural, and we turn the descriptor object itself rather than performing the attribute lookup. With the revised code we see more helpful behavior. If you need to query or manipulate the class which contains the descriptor object you can get hold of this through the owner argument of the dunder get method, which in this case, will contain a reference to the Planet class. In many cases, though, you won't need to use Owner, so you can do what we've done and just ignore it.**

**Data vs. Non-data Descriptors**

**You may here the terms data descriptor and non-data descriptor, but what do they mean? A non-data descriptor is a descriptor which implements only the dunder get method, and so is read only. So called data descriptors sport both the dunder get and dunder set methods and are writable. The distinction is important because of the precedence of attribute lookup. Behind the scenes the machinery in object. dunder getattribute, which is responsible for all attribute lookup in Python, transforms a call like this into a call like this. In fact, a precedence chain controls attribute lookup according to the following rules. If an instance is dunder dict has an entry with the same name as a data-descriptor the data-descriptor takes precedence. If an instance is dunder dict has an entry with the same name as a non-data descriptor the dictionary entry takes precedence. These statements are true, but quite a mouthful. Another way of looking at it is that attribute lookup precedes first the data descriptors, such as properties defined in the class, then to instance attributes in dunder dict, and then onto non-data descriptors in the class again. A simple experiment should make things clearer. Here we define a DataDescriptor, a NonDataDescriptor, and a class which uses them both called Owner. We'll import all this into a REPL session. After we've created an instance of Owner we'll retrieve the attribute A, set an item in the instance dictionary with the same name, and retrieve A again. Since this is a data descriptor the first rule applies, and the data descriptor takes precedence when we call obj. a. Now let's try attribute b. The first time we access obj. b there is no entry of the same name in the instance dictionary, so the NonDataDescriptor takes precedence. After we've added a b entry into dunder dict the second rule applies, and the dictionary entry takes precedence over the NonDataDescriptor.**

**Summary**

**Let's summarize. As we've seen, the descriptor protocol itself is very simple, which can lead to some very concise and declarative code, which hugely reduces duplication. At the same time, implementing descriptors correctly can be tricky, and requires careful testing. In this module we reviewed the built-in property decorator. We demonstrated how to create property descriptors without using decorator syntax by passing property getters and setters directly to the property constructor. These reinforce the notion that properties create objects called descriptors, which are bound to class attributes. We showed how to implement a simple descriptor to perform a basic attribute validation check by creating a class that implemented the descriptor protocol. We explained how descriptor instances have no knowledge of to which class attributes they are bound, so each descriptor instance must store the instance attributes for all descriptor owner instances. This can be achieved by using a WeakKeyDictionary from the Python standard library Weak Ref module. We looked at how to know when descriptors are retrieved from their owning classes rather than via instances by detecting when the instance argument to dunder get is set to none. A natural course of action in such cases is to return the descriptor instance itself. We explained the distinction between data and non-data descriptors, and how this relates to attribute lookup precedence. We're not quite done with the descriptors yet, and in particular, we'd like descriptor instances to know the name of the class attributes to which they have been bound, but to solve that problem we need sharper tools in the form of Python's meta classes, which we'll be covering next. Thanks for watching, and we'll see you in the next module.**

**Instance Creation**

**Instance Creation**

**Hello. My name is Robert Smallshire. Welcome to the fifth module of the Advanced Python course. In this course module we'll be taking a deep dive into exactly what happens when we create a new object. With this knowledge in hand we'll be able to exercise fine control over instance creation, which allows us to customize Python objects in powerful ways. So, what does happen when you create an object? Consider this simple class which models the coordinates on a chess board consisting of a file letter from a to h inclusive, and a rank number from one to eight inclusive. The class implements an immutable value type. The initializer establishes the invariance, and the property accessors prevent inadvertent modification of the encapsulated data. It's easy to think of dunder init as the construct implementation, but let's look closely at what happens when we create an instance of chess coordinate by calling the constructor. Here we create an instance called white\_queen at coordinate (d, 4). We'll step in with the debugger, pausing on the first line of dunder init. This much should already be evident from the signature of dunder init, which accepts self as its first argument, but the object referred to by self already exists. That is to say, the object has already been constructed, and the job of dunder init really is just to initialize the object. At this juncture our debugger is having difficulty displaying the uninitialized object because our dunder repr implementation, quite reasonably, expects that the object has been initialized. However, we can add a watch in the debugger for type self to see that self already has the required type, and self. dunder dict to see that the instance dictionary is empty. As we continue to step through the initializer we can watch as the instance dictionary is populated by assignments to attributes of self. Behind the scenes, object. dunder setattr is being called. Note also that dunder init doesn't return anything. It simply mutates the instance it has been given. So if dunder init isn't responsible for creating the instance, what is? If we look at the special methods of ChessCoordinate using a call to dir we can see one called dunder new. We haven't defined dunder new ourselves, but we do inherit an implementation from the universal base class object, so it is the base class implementation of dunder new, which is responsible for allocating our object in this case. A simple test demonstrates that ChessCoordinate. dunder new is in fact the very same method as object. dunder new, but what is the signature of dunder new? Don't bother looking in the help because, frankly, the answer isn't very helpful. We have all the answers though, and we'll be overriding dunder new to demonstrate.**

**Allocation with \_\_new\_\_()**

**So let's figure out the signature of dunder new. We'll implement the most basic override of dunder new, which simply delegates to the base class implementation, although we'll add a few print statements, so we can easily inspect the arguments and return value. Notice that dunder new appears to be implicitly a class method. It accepts cls as its first argument rather than self. In fact, dunder new is especially cased static method that happens to take the type of the class as its first argument, but the distinction isn't important here. The cls argument is the class of the new object, which will be allocated. In our case, that will be ChessCoordinate, but in the presence of inheritance it isn't necessarily the case that the cls argument will be set to the class in closing the dunder new definition. In general, dunder new accepts in addition whatever parameters have been passed to the constructor. In this case, we've soaked up any such arguments using \*args and \*\* keyword args, although we could have used specific argument names here, just as we have with dunder init. We'll print these additional argument values to the console. Remember that the purpose of dunder new is to allocate a new object. There's no special command for that in Python. All object allocation must be done by the dunder new implementation on the ultimate base class object. Here, rather than call the object. dunder new implementation directly, we'll call it via super. Should our immediate base class change in future this is more maintainable. The return value from the call to object. dunder new is the new instance of ChessCoordinate. We'll print its id. Remember, we can't expect repr to work yet, and then return. It is this returned object that is then passed as the self-argument to dunder init. We have printed the id of self here in dunder init to demonstrate that this is indeed the case. We see that the id of obj in dunder new is equal to the id of self in dunder init, and also that the constructor arguments have been forwarded to dunder new as expected.**

**Customizing Allocation**

**Now let's look at customizing allocation by overriding dunder new. We've shown the mechanics of overriding dunder new. We accept the class type and the constructor arguments and return an instance of the correct type. Ultimately, the only means we have of creating new instances is by calling object. dunder new. This is all well and good, but what are some practical uses? One use for controlling instance creation is a technique called interning, which can dramatically reduce memory consumption. We'll demonstrate by extending our program to allocate some chess boards in the start of game configuration. In our implementation each board is represented as a dictionary mapping the name of the piece to one of our ChessCoordinate objects. For fun we've used Unicode chess code points to represent our pieces. In our program a string comprised of white queen and white rook means white queen's rook, and a string composed of black king, black bishop, black pawn means black king's bishop's pawn. We need to be this specific because, remember, dictionaries require that keys are distinct. Our revised program now has a starting board function and main creates, for the time being, a single search board. We've also removed the various print statements from our dunder new and dunder init functions. If we create a single chess board this way a peek in activity monitor on our Mac shows that Python is allocated about 17. 1Mb of memory. Creating 10, 000 chessboards utilizes some 84. 2Mb of memory to store the 320, 000 instances of ChessCoordinate contained by the 10, 000 dictionaries that must have been created to represent all the boards. Bear in mind, though, that there are only 64 distinct positions on a chess board, and given that our ChessCoordinate objects are deliberately immutable value types, we should never need more than 64 instances. In our specific case, we should never need more than the 32 positions occupied by the pieces in their initial places. Let's put in place updated definitions for dunder new and dunder init, which achieve just that. We're now using named position arguments for the file and rank arguments to dunder new, and we'll move the validation logic from dunder init to dunder new. Once the arguments are validated we use them to create a single tuple object from file and rank to use as a key, and check if there is an entry against this key tuple in a dictionary called \_interned we've attached as a class attribute. Only if the tuple key is not present in the dictionary do we allocate a new instance by calling object. dunder new. We then configure the new instance during the remainder of the work that used to be done in dunder init, and insert the newly minted instance into the dictionary. Of course, the instance we return is whichever instance we have just or previously inserted into the dictionary. Our dunder init method is now empty and can, in fact, be removed entirely. With these changes in place allocating 10, 000 boards takes much less memory than previously. In fact, we're down to 37. 1Mb, less than half what we required previously. Remember that even though we've interned the ChessCoordinate type we still have to store 10, 000 distinct dictionaries representing the board configurations. Interning is a powerful tool for managing memory usage. In fact, Python uses it internally for integers and strings, but it should only be used for immutable value types, such as our ChessCoordinate where instances can safely be shared between data structures.**

**Summary**

**We've covered a crucial topic in this short course module, the distinction between the allocation and initialization of instances. We showed how the static method, dunder new, is called to allocate and return a new instance. It is implicitly a static method, which accepts the class of the new instance as its first argument, and doesn't require either the class method or static method decorators. Ultimately, object. dunder new allocates all instances. One use for overloading dunder new is to support instance interning, which can be useful when certain values of immutable value types are very common or when the domain of values is small and finite, such as with the squares of a chess board. This isn't the whole story though, and Python offers yet more control over instance creation at the class level, although before we get to that we need to understand meta classes. Thanks for watching, and we'll see you in the next module.**

**Metaclasses**

**Metaclasses**

**Hello. My name is Robert Smallshire. Welcome to the sixth module of the Advanced Python course. We've mentioned metaclasses several times. We owe you an explanation. In this module we'll look at the concept of the class of class of objects, metaclass for short, the default metaclass called type, specifying metaclasses in class definitions, defining metaclasses, including the special methods of metaclasses, some practical examples of metaclasses, which solve some problems we've discovered earlier in the course, and a look at how metaclasses interact with inheritance. To assist us on our journey to understand metaclasses we need a simple class. We'll use Widget, which will be empty. We'll instantiate some widgets and introspect their types using the type built-in function. We all know that in Python the type of an instance is its class, so what is the type of its class? The type of a class is type. We've covered this before in the introspection module of Python Beyond the Basics, but it bears repeating here. One potentially confusing aspect here is that we're using type as a function to determine the type of an object, but type is also being used as a value. It is the type of the class object. However, this duality isn't so unusual. We see exactly the same situation with, say, list, where it is used as both a constructor and the value. Here we use the list constructor to construct a list A, and the type of a is also list. Given that the type of the Widget class is type, we can say that the metaclass of a widget is type. In general, the type of any class object in Python is its metaclass, and the default metaclass is type. Going even further, we can discover that the type of type is type. We can get the same results by drilling down through the special attribute, dunder class. Here's w. dunder class, w. dunder class dunder class, and w. dunder class dunder class dunder class, which is the base of the recursion. To understand where metaclasses fit into Python we must consider how class objects are created. When we define a class in a Python source file like this, with class Widget, this is actually shorthand for class Widget with the object base class and the type metaclass. In the same way that the base class is implicitly object, the metaclass is implicitly type.**

**Class Allocation and Initialization**

**Roughly speaking, when we write a class block in our Python source code it is syntactic sugar for creating a dictionary. The Python runtime populates the namespace dictionary while reading the contents of the class block, which is passed to a metaclass to convert the dictionary into a class object. To see how that works it's important to understand the several tasks that metaclasses perform during creation of a new class. When we write class Widget what's actually happening is something like this, which is to say that the class name is Widget, the metaclass is type, the class has no base classes, other than the implicit object, no keyword arguments were passed to the metaclass, we'll cover what this means later. The metaclasses, dunder prepare method is called to create a new namespace object, which behaves like a dictionary. Behind the scenes the Python runtime populates the namespace dictionary while reading the contents of the class block. The metaclasses dunder new method is called to allocate the class object, and finally, the metaclasses dunder init is called to initialize the class object. The name, bases, and namespace arguments contain the information collected during execution of the class definition, normally the class attributes and method definitions inside the class block, although in our case, the class block is logically empty. By providing our own metaclass we can customize these behaviors. We'll start with a very simple metaclass called TracingMeta in a module, tracing. py, which simply prints its method invocations and return values to the console for each of dunder prepare, dunder new, and dunder init. Notice that our metaclass should be a subclass of an existing metaclass, so we'll subclass type. Each of our overrides delegates to the base class type method to do the actual work required via a call to super. Notice that although dunder new is implicitly a class method, we must explicitly decorate the dunder prepare class method with the appropriate decorator. Now we'll define a class containing a simple method called action, which just prints a message, and a single class attribute, the answer, with the cosmically inevitable value 42. We'll do this at the REPL, so you can see clearly when the metaclass machinery is invoked. At the instant we complete the class definition we can see from the tracing output that Python executes the dunder prepare, dunder new, and dunder init method in turn. First of all, let's look at dunder prepare, the purpose of which is to produce an initial mapping object to contain the class namespace. The mcs argument, short for metaclass, is a reference to the metaclass itself. This first argument is analogous to the self-argument passed to instance methods, and the cls argument passed to class methods. For metaclass methods it is conventionally called mcs. The name argument contains the name of our Widget class as a string. The bases argument is an empty tuple. We didn't declare any base classes for widget, and the ultimate object based class is implicit. The kwargs argument is an empty dictionary. We'll cover the significance of this shortly. The most important aspect of dunder prepare is that when it calls it's super class implementation in type the return value is a dictionary or, more generally, a mapping type. In this case, it's a regular, empty dictionary. This dictionary will be the namespace associated with the nascent class. Moving on to looking detail at dunder new, the purpose of which is to allocate the new class object, the mcs argument is a reference to the metaclass as before. The name and bases arguments are still the string name of the new class and the tuple of base classes. The mapping object we returned from dunder prepare is passed as the namespace argument to dunder new. The Python runtime has populated this dictionary with several entries, as it has processed the class definition of widget. Two of the items are our action method and the the answer class attribute. The other two items are dunder module and dunder qualname, which the Python runtime has added. The dunder module attribute is mapped in the name of the module in which the class was defined. Because we use the REPL this is built-ins. The dunder qualname attribute contains the fully qualified name of the class, including parent modules and packages. In this case, it just contains the class name, as the built-in's module used by the REPL is available everywhere, being the last namespace in the LEGB lookup hierarchy. The kwargs dictionary passed to dunder new is also still empty. Within dunder new we delegate to the base class, type. dunder new, via a call to super, forwarding the mcs, name, bases, and namespace arguments. The object returned by this call is the new Widget class. We are in the process of allocating and configuring. The new Widget class is what we returned from dunder new. Note that any changes we wish to make to the contents of the namespace object must have been made before this call, as this is the point at which the class object is created. To change the contents of the class namespace after this call the class object must be manipulated directly. Finally, we come to dunder init, the purpose of which is to configure the newly created class object. Note that dunder init here is an instance method of the metaclass, not an explicit class method like dunder prepare or an implicit class method like dunder new. As such, it accepts cls as its first argument, which is one level less meta than cls. The name, bases, namespace, and kwargs arguments are all as before. Again, we delegate to the type base class via super, although dunder init doesn't return anything it's expected to it's expected to modify the existing class object that was handed to it. Note that although the namespace object is passed to dunder init, its content should already have been used upstream by dunder new when allocating the class object. Changes to namespace will be ineffectual, and any changes to the class object must be affected by manipulating cls directly. The key here is that metaclasses give us the opportunity to modify the dictionary of class attributes, which includes methods, before the class is instantiated. We even get the opportunity to modify the list of base classes or produce an entirely different class, if required, although such uses are rare. We'll look at some complex examples soon to make this clear. You may be wondering which out of dunder prepare, dunder new, and dunder init you should override. If you don't override dunder prepare the default implementation in type will produce a regular dictionary for the namespace object, so you only need to override it if you need the behavior provided by another mapping type. Usually it will only be necessary to override either dunder new or dunder init, but of these two only dunder new can make decisions before the new class is allocated. The distinction between dunder new and dunder init for metaclasses is exactly the same as it is for regular classes. Later we'll see that it might be wise to prefer a configuration in dunder init rather than dunder new, so that metaclasses are more composable.**

**Metaclass Keyword Arguments**

**Now let's look at a little known feature of Python which allows us to pass additional arguments to the metaclass. Earlier we skipped over the fact that dunder prepare, dunder new, and dunder init can all support a kwargs parameter to accept arbitrary keyword arguments. These can be supplied in the parameter list when defining a class, and any key with arguments provided over and above the named metaclass argument will be forwarded to these three special methods. Here's an example where we've passed a tension keyword argument with a value of 496 in the parameter list of our Reticulator class definition, which uses TracingMeta as its metaclass. As you can see, the arguments are dutifully forwarded to the metaclass methods, and the argument value could have been used to configure the class object. This allows the class statement to be used as a kind of class factory. Here's an interesting example, which uses keyword args. The dunder new method of the entries meta metaclass expects kwargs to contain a num\_entries key, which maps to an integer value. This is used to populate the namespace with named entries using the letters of the alphabet. At first sight it looks like we only need to override dunder new to achieve our aims. Let's try to use it. We'll import EntriesMeta from entries, and then define a new class, AtoZ, with EntriesMeta as its metaclass, and passing the custom argument, num\_entries, as 26 to get the 26 letters of the English alphabet. The trace output shows that num\_entries 26 is successfully passed to dunder new, but later we get an error from dunder init. The problem we've set up deliberately here is that both dunder new and dunder init must accept any additional keyword arguments. They must have the same signature. We need a do nothing dunder init to keep Python happy. Here it is. With this important change in place everything works as intended. We import EntriesMeta, define AtoZ with EntriesMeta and num\_entries 26. We see the num\_entries item arriving keyword args, and we can see the namespace object populated with all the additional entries. Of course, we can just use a regular argument, as well as or instead of double star keyword args. Let's convert num\_entries into a proper argument, in addition to double star keyword args. This makes the code much easier to understand, and still works exactly as before.**

**Metaclass Method Visibility**

**Now let's look at metaclass methods and their visibility. We've covered three important special methods for metaclasses; dunder prepare, dunder new, and dunder init, but what happens if we include other methods in the metaclass? Let's see by adding a method called metamethod to the TracingMeta metaclass we were experimenting with earlier. It accepts a single argument, cls, which remember, means it is an instance method of the metaclass. Within it we simply print its argument. Let's create the class widget again with its TracingMeta metaclass. As we complete our Widget class definition the metaclasses are invoked. It turns out that instance methods of the metaclass can be accessed similarly to class methods of the Widget class, and will have the regular class passed to them as the first and implicit argument. Here we call metamethod directly on the Widget class. However, unlike regular class methods we create with the class method decorator, we cannot access so called metamethods via the Widget instance. Metamethods are rarely used in practice, although one metamethod in particular has an interesting use, dunder call, which we'll look at shortly. Class method definitions in the regular class and its base classes will take precedence over looking up a method in its metaclass. Regular methods of the metaclass accept cls as their first argument. This makes sense because cls, the class, is the instance of the metaclass. It is analogous to self. On the other hand, class methods of the metaclass accept mcs as their first argument, the metaclass, analogous to the cls argument of a class method in a regular class.**

**Metaclass \_\_call\_\_ : The Instance Constructor**

**Now let's look at how to achieve very fine-grained instantiation control by overriding metaclass dunder call. For a moment return your thoughts to the material we covered in the previous module of this course when we looked at instance allocation with dunder new. We know that in order to create instances we call the constructor of the designed class. We have learned that behind the scenes this will call Widget. dunder new to allocate a widget followed by Widgit. dunder init, to do any further initialization. Let's pull back the curtain, and see exactly what is behind the scenes. The behavior of calling dunder new followed by dunder init when we call a constructor is actually the responsibility of dunder call on the metaclass. This makes sense when we remember that dunder call is a metamethod, and therefore can be called like a class method, and that dunder call makes the objects on which it is defined callable, like functions. This is the mechanism by which classes in Python become callable, and what we have been referring hither to as a constructor call is, in fact, the dunder call metamethod. Let's see this in action by overriding dunder call in our TracingMeta example. Within dunder call we print out its name and arguments, forward the call with super to the type metaclass, and then print out the return value from the super class. We'll also implement a TracingClass, which will use TracingMeta as its metaclass. Tracing class overrides dunder new and dunder init, so we can see when they're called. Notice that when we import the module into the REPL the metaclass trifecta, dunder prepare, dunder new, and dunder init are invoked when the TracingClass is defined. Now we'll instantiate TracingClass with a positional argument and a keyword argument to its constructor, 42 and keyword clef. Look carefully at the control flow here. Our call to the constructor invokes dunder call on the metaclass, which receives the arguments we passed to the constructor, in addition to the type we're trying to construct. Out dunder call override calls the super class implementation, which is type. dunder call in this case. See how type. dunder call in turn calls TracingClass. dunder new, followed by TracingClass. dunder init. In other words, it is type. dunder call, which orchestrates the default class allocation and initialization behavior. It is very rare to see the dunder call metamethod overridden. It's pretty low level in Python terms, and provides some of the most basic Python machinery. That said, it can be powerful. In keywordmeta. py we have an example of a metaclass overriding dunder call, which prevents classes, which use the metaclass, excepting positional arguments to their constructors. In dunder call we detect if args is non-empty, and if that's the case we raise a type error. We've also defined a regular class called ConstrainedToKeywords, which uses KeywordsOnlyMeta as its metaclass. Even though the dunder init method in ConstrainedToKeywords accepts positional arguments through \*args, execution never gets this far, as non-empty positional argument lists are intercepted by dunder call in the KeywordsOnlyMeta metaclass, which causes a type error to be raised. Constructor calls which contain only keyword arguments are permitted, as we designed. We've covered a lot of the theory and practice behind metaclasses. Now well build on those ideas with some useful applications.**

**A Practical Metaclass Example**

**Python metaclasses can seem very, well, meta, so it's time for a session on metaclasses, which solve some actual problems you've probably encountered. At some point you've probably run into the issue that duplicate class attribute names aren't flagged by Python as errors. Here we have a class, Dodgy, which contains two definitions for method. In fact, the second definition takes precedence because it overwrites the first entry in the namespace dictionary as the class definition is processed. Let's write a metaclass which detects and prevents this unfortunate situation occurring. To do this, rather than using a regular dictionary as the namespace object used during class construction, we need a dictionary which raises an error when we try to assign to an existing key. Here is such a dictionary, OneShotDict, which is implemented by specializing the built-in dict type and overriding the dunder init and dunder set item methods. Note that the built-in dict has a quite sophisticated initializer, which accepts many forms of arguments, but something much simpler is sufficient in our case. After calling the super class dunder init the dunder init implementation loops over any entries which have been passed, inserting each key and value pair in turn by call to dunder setitem. In dunder setitem we check whether the key is already in the dictionary, and if it is we raise a value error. Before we use the OneShotDict in a metaclass let's just check that it's working as expected. We'll create an instance of OneShotDict called d, insert the value 65 against the key A, insert the value 66 against the key B, and then try to insert the value 32 against the key A. As hoped, the value error is triggered. Now we can design a very simple metaclass, which uses OneShotDict for the namespace object called ProhibitDuplicatesMeta. All we need to do is override the dunder prepare class method, returning an instance of our specialized dictionary. Now if we try to define a class with duplicate methods using this metaclass we get an error. The main shortcoming here is that the error message isn't hugely informative. Unfortunately, we don't have access to the part of the runtime machine, which reads our class definition and populates the dictionary, so we can't intercept the ValueError and emit a more useful error instead. The best we can do, rather than using a general purpose collection, like OneShotDict, is simply to create a functional equivalent called something like OneShotClassNamespace with a more specific error message. This has the benefit that we can pass in additional diagnostic information, such as the name of the class currently being defined into the namespace object on construction, which helps us produce a more useful message. We adjust its initializer to accept a positional name argument, which we store as an instance attribute, \_name. In the guard clause of dunder set item we change the exception type from value error to type error, and edit the error message to make it both more germane and more informative. Lastly, we need to remember to forward the class name, which is past to dunder prepare, to the OneShotClassNamespace constructor. When we try to execute the module containing the class with the duplicate method definition we get a much more useful error message, cannot reassign existing class attribute 'method' of 'Dodgy'. Much better.**

**Naming Descriptors Using Metaclasses**

**For our next metaclass example let's return to the Planet example we used in a previous course module to illustrate descriptors. Here's a reminder of the code we ended up with. Recall that we implemented a new descriptor type called Positive, which would only emit positive numeric values. This saved a lot of boilerplate code in the definition of our Planet class, but we lost an important capability along the way because there is no way for a descriptor instance to know to which class attribute it has been bound. One of the instances of positive is bound to Planet. radius\_metres, but it has no way of knowing that. The default Python machinery for processing class definitions just doesn't set up that association. The shortcoming is revealed when we trigger a value error by trying to assign a non-positive value to one of the attributes. Here we try to give the planet Mercury a nonsensical negative mass of -10, 000. The error message doesn't, and in fact, can't tell us which attribute triggered the exception. Now we'll show how we can modify the class creation machinery by defining a metaclass which can intervene in the process of defining the Planet class in order to give each descriptor instance the right name. We'll start by introducing a new base class for our descriptors called Named. This is very simple and just has name as a public instance attribute. The constructor defines a default value of none because we won't be in a position to assign the attribute value until after the descriptor object has been constructed. We'll modify our existing positive descriptor, so it becomes a subclass of named, and therefore gains the name attribute. Again, the constructor arguments define a default of none. We've modified the argument listed, dunder init, ensured that the super class initializer is called, and made use of the new name attribute in the error messages raised by dunder set and dunder delete. Now we need a metaclass which can detect the presence of descriptors which are named, and assign the class attribute names to them. Here is DescriptorNamingMeta. Again, this is fairly straightforward. In dunder new we iterate over the names and attributes in the namespace dictionary, and if the attribute is an instance of named we assign the name of the current item to its public name attribute. Having modified the contents of the namespace we then call the super class implementation of dunder new to actually allocate the new class object. The only change we need to make to our Planet class is to refer to the metaclass on the opening line. There's no need for us to modify the uses of our positive descriptor. The optional name argument will default to none when the class definition is first read before the metaclass dunder new is invoked. By trying to set a non-positive mass for the planet Mercury we can see that each descriptor object now knows the name of the attribute to which it has been bound, so it can emit a much more helpful diagnostic message, such as Attribute value mass\_kilograms -10000 is not positive, allowing us to much more easily track down the problem.**

**Metaclasses and Inheritance**

**We'll finish off this module by looking at how metaclasses interact with inheritance. Let's define two metaclasses, MetaA and MetaB, related only by the fact that they both subclass type. Both metaclasses will be empty. We'll also define two regular classes, A and B, which use MetaA and MetaB as their respective subclasses. Now we'll introduce a third class, D, which derives from class A. Now let's introspect class D itself, not an instance of D with type D to determine what its metaclass is. The metaclass of D is MetaA, which was inherited from its regular base class A, so metaclasses are inherited, but what happens if we try to create a new class C, which inherits from both regular classes A and B with their different metaclasses, MetaA and MetaB? Let's give it a go. When we try to execute this code by importing it at the REPL we get a type error with the message, metaclass conflict: the metaclass of a derived class must be a (non-strict) subclass of the metaclasses of all its bases. Python is telling us it doesn't know what to do with the unrelated metaclasses, MetaA and MetaB. Which metaclass dunder new should be used to allocate the class object? To resolve this we need a single metaclass. Let's call it MetaC, which we can create by inheriting from both MetaA and MetaB. In the definition of regular class C we must override the metaclass to specify MetaC. With these changes we can successfully import C and check that its metaclass is MetaC by asking for the type of C. So we've persuaded Python to accept our code, but our metaclasses are empty, so combine trivially. Sometimes metaclasses will combine in straightforward ways. For example, our ProhibitDuplicatesMeta, which overrides only dunder prepare, and our KeywordsOnlyMeta, which overrides only dunder call, can be combined into the conceptually simple, but horribly named, ProhibitDuplicatesAndKeywordsOnlyMeta, which can have an empty class body, but combine both behaviors. To cooperate gracefully non-trivial metaclasses must be designed with this in mind, which isn't always straightforward or even possible if the combination makes no sense. An important step in designing cooperative metaclasses is to diligently use super when delegating to base classes because, as we learned in Python Beyond the Basics, super actually delegates to the next class in the method resolution order or MRO, which accounts for multiple inheritance. Even though both our TracingMeta class we defined earlier, and the DescriptorNamingMeta metaclass we used with the Planet attributes both override dunder new, they combine easily in either order because our implementations delegate to the next class in the MRO chain using calls to super. In planet. py we can import TracingMeta, and combine our two metaclasses into TracingDescriptorNamingMeta with multiple inheritance. The planet class definition is executed when we import the module. This is when the metaclasses do their work, and we can see that tracing works as expected. We can also confirm that the descriptor naming is working by causing a value error and checking for the descriptor name in the error message. The two metaclasses cooperate successfully by using super. Another tip for designing metaclasses which cooperate well is to prefer to put code which configures the class object in the dunder init metamethod, which is handed the class object to be configured rather than the dunder new metamethod, which is responsible for allocating the new class object. Of course, each object may be only allocated once, but could be configured multiple times. Ultimately though, if you're dependent on metaclasses from a third party framework, such as SQLAlchemy or Qt, there's a good chance their metaclasses won't compose gracefully. You should be able to figure out whether they will compose by reading their source code if you have access to it.**

**Summary**

**We've covered a lot of ground in this course module, and you should now know more than the majority of Python developers about the customization of class creation using metaclasses. All classes have a metaclass, which is the type of the class object. The default type of class objects is type. The metaclass is responsible for processing the class definition parsed from the source code into a class object. The dunder prepare metaclass method must return a mapping object, which the Python runtime will populate with the namespace items collected from parsing and executing the class definition. The dunder new metaclass method must allocate and return a class object and configure it using the contents of the class namespace, the list of base classes passed from the definition and any additional keyword arguments passed from the definition. The dunder init metaclass can be used to further configure the class object and must have the same signature as dunder new. The dunder call metaclass method, in effect, implements the constructor for instances of the class, and is invoked when we construct an instance. An important use case for metaclasses is to support so-called named descriptors, whereby we can configure descriptor objects, such as properties, with the name of the class attribute to which they are assigned. Strict rules control how multiple metaclasses interact in the presence of inheritance relationships between the regular classes which use them. Judicial metaclass design using super to delegate via the MRO can yield metaclasses which compose gracefully. Thanks for watching, and we'll see you in the next module where we'll look at a simpler, but less capable alternative to metaclasses for many uses, class decorators.**

**Class Decorators**

**Class Decorators**

**Hello. My name is Robert Smallshire. Welcome to the seventh module of the Advanced Python course. There's no doubt that metaclasses can be difficult to understand and reason about. Fortunately, Python supports an alternative, which is sufficient for many cases where we want to customize classes at the point they are defined. This alternative is the class decorator. It's worth bearing in mind that anything that can be achieved with a class decorator can also be achieved with a metaclass, although the reverse is not true. In other words, class decorators are less powerful than metaclasses, although they are much easier to understand, so should be preferred whenever the desired effect can be achieved with either a metaclass or a class decorator. Class decorators work in much the same way as function decorators. They apply a transformation to a class after the class definition body has been processed, but before the definition is bound to the name of the class. Let's start with a very simple temperature class, which is decorated with a function, my\_class\_decorator. Set aside for the moment that these getter and setter methods are hardly the most Pythonic solution, they'll serve our purpose for the time being. To understand how class decorators work, my class decorator is a very simple function, which simply iterates over and prints the attributes of the class object. The class decorator must accept the class object as its only argument by a convention called cls, and must return the modified class object or an alternative class object, which will be bound to the class name given in the definition, in this case, Temperature. When we import our module into a REPL session we can see that the decorator is executed when the temperature class is first defined, which is when the module is first imported. The class attributes discovered by our decorator are dunder module, our get\_kelvin and set\_kelvin methods, dunder init, dunder weakref, and dunder dict. It's important for our Temperature class to maintain an important invariant of the universe in which we live that temperatures cannot be lower than absolute 0, which is 0 Kelvin. Rather than require that every method of our class individually honors this class invariant, we'll use a class decorator to wrap every method in an invariant checking proxy.**

**Enforcing Class Invariants**

**To do this in a generic way we need to provide a way of specifying the class invariant to the class decorator. Recall from our earlier work with function decorators, in Python Beyond the Basics, to do this we define a function accepting the arguments we need, which in turn returns a decorator. A predicate function which describes the invariant will be passed to a factory function, a class decorator factory, if you will, which creates the actual class decorator, which processes the definition of the temperature class. As the temperature class is processed by the class decorator each callable member of the class is identified in turn. These are the methods of the temperature class, and a function decorator is applied to each method. When invoked, the method decorators delegate to the original predicate function to verify the invariant. There's a lot going on here, so let's look at the code. The invariant function, which is, in effect, a decorator factory, accepts a predicate function, which is used to test the invariant, and returns the invariant\_checking\_class\_decorator function. The class decorator accepts the class object as cls and builds a list of method names by identifying the callable attributes. Each method is then processed by a function called wrap\_method\_with\_invariant\_checking\_proxy, which creates a function decorator for each method, which calls the method, and then checks that the invariant predicate still holds. Note that we must use setattr to update the class namespace, rather than manipulating the class dunderdict directly, as the latter is not mutable. We can now define our temperature class like this, calling the invariant class decorator function and passing to it our not\_below\_absolute\_zero predicate function. Our predicate function is defined as our free function, not\_below\_absolute\_zero, which simply tests the \_kelvin private attribute. The predicate function necessarily directly accesses the private attributes of the temperature class. That's fine. The class and the function are very closely related. Creating non-negative temperatures works as expected. Here we create a temperature of 5 Kelvin, but an attempt to construct a negative temperature, here we tried to create a temperature of -1 Kelvin, demonstrates that our class decorator has successfully wrapped dunder init with the function decorator which checks the invariant. Likewise, the function decorator successfully wraps set Kelvin. After constructing a temperature of 42 Kelvin we tried to modify the temperature to -1 Kelvin, and the runtime error is raised. Before we get too excited, let's see what happens if we introduce some properties, defined using the property decorator into our Temperature class. We've added properties to get and set the temperature in both Celsius and Farenheit. We create a temperature of 42 Kelvin, which is reported back to us by the Celsius getter as about -231 Celsius. This is fine. Now let's set the Celsius to -100. That's fine because -100 Celsius is well above absolute zero. This works without a hitch, and the temperature is reported back to us in Celsius accurately. Now let's try to set a temperature of -300 Celsius, which is well below absolute 0. At this point, we should have received a runtime error, as -300 Celsius is less than absolute 0 Kelvin; however, no exception is raised, and the class invariant has been violated. If we try to get the Kelvin value via the get\_kelvin method we do indeed get an error, but this is too late. Class invariant violations should never be detected by non-mutating getters or other query methods. A breach in our defenses has permitted the object to get into an invalid state. How is our invariant check being circumvented?**

**Detecting and Wrapping Properties**

**Recall that the property decorator produces a descriptor, which wraps the getter and setter methods, but because descriptors aren't callable functions they aren't being detected and wrapped by our invariant checking function decorator. To fix this we need to beef up our machinery to detect and appropriately proxy these descriptors. If we take a peek at vars(Temperature) we can see that Fahrenheit and Celsius correspond to property objects. Ideally, we detect any value in this mapping, which is a descriptor, which is any object supporting any of the dunder get, dunder set, or dunder delete methods. Unfortunately for us, in this case, descriptors are also used for all the functions and methods for reasons we won't even get into in this advanced course, so they too would be detected. Instead, we'll go for the simpler approach of looking for instances of property, how updated class decorator factory function looks like this. After we have detected the method names by looking for callable attributes, we detect the property names by looking for attributes where the value is an instance of property. We'll insert a new section into the body of the class decorator itself to search for and wrap any class attributes which are properties. This delegates to a new function called wrap\_property\_with\_invariant\_checking\_proxy. This function retrieves the property from the class, and passes it and the predicate function, to the constructor of a new class, InvariantCheckingPropertyProxy. We use setattr to replace the property with the proxy. The InvariantCheckingPropertyProxy is itself a descriptor, implementing dunder get, dunder set, and dunder delete. The initializer simply stores references to the referent property and the predicate function. The implementations of dunder get, dunder set, and dunder delete forward to the underlying referent property, and then check that the class invariant of the instance remains unviolated. A quick test at the REPL shows that everything is working as designed. We'll create a Temperature of 42 Kelvin, try to use the Celsius setter to set a temperature of -300 Celsius, which results in a runtime error, showing that our InvariantCheckingPropertyProxy is working. A temperature of 100 Fahrenheit is admissible, and we can convert that to Celsius. Everything seems to be working fine.**

**Chaining Class Decorators**

**For a final demonstration we'll show that we can chain class decorators just as we can with function decorators. We've added a second invariant to be maintained to ensure that the temperature is below the hypothetical value, absolute hot. You can read more about this on Wikipedia if you're interested in the physics. At the REPL we can see that both constraints are enforced when we call instance methods, such as set to Kelvin. We start with a temperature of 37. 5 Kelvin, we're prevented from setting a negative Kelvin value, and if we try to set a Kelvin value larger than absolute hot, 1 \* 10 to the 33, the attempted invariant violation is detected. Our class decorator has no problem decorating the already decorated methods on its second implication. The proxy and properties are another story though. Although the lower bound check works as before on the properties, when we try to set the temperature of -300 the violation is detected. Attempts to violate on the upper bound are not detected, so setting a temperature of 1 \* 10 to the 34 Celsius doesn't fail as it should. The problem here is that our class decorator is detecting specifically property instances with the is instance check; however, our invariant checking and property proxy is not a property, so our proxy which enforces the not\_below\_absolute\_0 invariant is applied to the genuine property, although the below\_absolute\_hot proxy is not applied to the invariant checking and property proxy. For a solution to this problem we'll use Python's abstract base class mechanism, a topic we'll explore in the next module of this course.**

**Summary**

**Let's summarize. Class decorators provide a simple alternative to metaclasses for processing class definitions prior to the definition being bound to the class name. Remember, though, that class decorators are less powerful than metaclasses. Use class decorators when you can and metaclasses when you must. Thanks for watching, and we'll see you in the next module on abstract base classes.**

**Abstract Base Classes**

**Introducing Abstract Base Classes**

**Hello. My name is Robert Smallshire. In this course module we'll be investigating the abstract base class mechanism in Python as originally defined in PEP 3119. We previously used some abstract base classes, such as Sequence from the collections. abc module in our Python Beyond the Basics course when implementing a sorted set collection type. This time we'll look at the tools provided in the Python standard library ABC module for creating abstract base classes of your own design. If you're coming to Python from another object oriented language, such as Java, C++, or C# you may have preconceived ideas of what an abstract base class is and how to use one. Beware though, the abstract base class mechanism in Python is much more flexible and can work in what may seem to be very surprising ways, so pay attention. Thinking for a moment beyond the confines of Python, what is, in general, an abstract base class? The clue is in the name. Base refers to the fact that the class is intended to be the target of an inheritance relationship. That is, we expect another class to derive from the base class. For instance, GraphicalDevice here is intended as the base class for other classes, such as our EpsonWF3520PrinterDriver. Abstract refers to the fact that the class cannot be instantiated in isolation. That is, it makes no sense to create an object of the type of the base class alone. It only makes sense to instantiate the class as part of an object of a derived type. Ideally, it should not be possible to instantiate an abstract base class directly. The opposite of abstract is concrete, and in this example the PrinterDriver is a concrete class, so it makes sense to instantiate it. The rational for any abstract base class is to define an interface which derived classes must implement. This allows client code to be written against the base class interface. In this example the PrinterDriver must override the three abstract methods of GraphicalDevice. Done diligently, this leads to a highly desirable property of class hierarchies called Liskov Substitutability, a design principle which states that subclasses, from the point of view of the client code, should be interchangeable. In other words, client code developed against an abstract interface should not require knowledge of specific concrete types, only of the capabilities as promised by the abstract base class. For example, code written against the interface of GraphicalDevice should be able to render to an EpsonWF3520Printer or a 1080p LcdDisplay without modification. We can substitute one concrete class for another. Abstract base classes differ from pure interfaces, such as those we have in languages like Java, insofar as they can also contain implementation code, which is to be shared by all derived classes. Why do we need to define named interfaces when we have duck typing? Isn't it sufficient to know whether a particular object responds to the interface of a duck, and behaves like a duck without actually knowing that it is a duck? In Python this is true both in theory and practice, but determining whether a particular object supports the required interface in advance of exercising that interface can be quite awkward. For example, what does it mean in Python to be a MutableSequence? We know that list is a MutableSequence, but we can't assume that all mutable sequences are lists. In fact, the MutableSequence protocol requires at least 16 methods are implemented. When relying on duck typing it can be difficult to be sure that you've met the requirements, and if clients do need to determine whether a particular object is admissible as a mutable sequence with a look before you leap approach the check is messy and awkward to perform robustly.**

**Abstract Base Classes in Python**

**Abstract base classes in Python serve two purposes. First of all, they provide a mechanism for defining protocols or interfaces, and ensuring the implementers of those protocols meets some minimum requirements. Secondly, they provide a means for easily determining whether an arbitrary class or instance meets the requirements of a specific protocol. For instance, after importing MutableSequence from the collections. abc module. We can determine that list is a MutableSequence using the built-in, issubclass function. This much may not be surprising, but let's look at the base class of list. In fact, we'll look at the transitive base classes of list by examining its method resolution order using the dunder mro attribute. This reveals that list has only one base class, object, and that MutableSequence is nowhere to be seen. Further reflection, if you'll excuse the pun, might lead you to wonder how it is that such a fundamental type as Python's list can be a subclass of a type defined in a library module. We've started out with this curious example, so as to efficiently disabuse migrants from other programming languages of any existing notions of what abstract base classes are, how they work, how they are used, and why they are useful. That done, we'll dig further into the mechanism. Let's establish that MutableSequence is indeed abstract by attempting to directly instantiate it. This fails with a useful type error explaining the five methods we need to implement the MutableSequence protocol. The reason we don't need to implement all 16 is that 11 of them can be implemented in terms of the other 5, and the MutableSequence abstract base class contains code to do exactly this. Note, however, that these implementations may not be the most efficient. Since they can't exploit knowledge of the concrete class, they must work entirely through the interface of the abstract class. Now to the question of how issubclass(list, MutableSequence) returns true. What happens is that when we call issubclass(list, MutableSequence) the built-in issubclass function checks for the existence of a method called dunder subclasscheck on the type of MutableSequence, which is to say, on the metaclass of MutableSequence, and if this method is present it is called with the subclass list as an argument. Issubclass(list, MutableSequence) is roughly equivalent to this code. It is up to the metaclass of MutableSequence to determine whether or not list is a subclass of MutableSequence, rather than list being required to know that MutableSequence is one of its base classes. This, in effect, allows list to be a subclass of MutableSequence without MutableSequence being a superclass of list in the normal sense of being the target of an inheritance relationship. We describe such base classes as virtual base classes, which has nothing at all to do with an identically named concept in C++. The dunder subclass check method on the metaclass of the virtual base class can do pretty much anything it likes to determine whether it's argument is to be considered a subclass.**

**Abstract Base Classes in Practice**

**Consider the code in weapons. py. In this module we've defined a Sword class with a metaclass, SwordMeta. SwordMeta defines the dunder subclasscheck method to check for the existence of callable swipe and sharpen attributes on the class. In this situation Sword will play the role of a virtual base class. A few simple tests at the REPL confirm that BroadSword and SamuraiSword are indeed considered subclasses of Sword, even though there is no explicit relationship through inheritance, so BroadSword is a subclass of Sword, SamuraiSword is a subclass of Sword, but Rifle is not a subclass of Sword. This isn't the whole story though, as tests of instances rather than classes using isinstance will return inconsistent results. Here we create an instance of SamuraiSword and try isinstance(samurai\_sword, Sword). This is false. This is because the isinstance machinery checks for the existence of the dunder instancecheck method, which we have not yet implemented. Let's do so now. Our dunder instance check implementation simply delegates to dunder subclasscheck, which is called as a metamethod on the actual class, cls. With this change in place our call to isinstance produces a result consistent with the result from issubclass. This surprising technique is used in Python for some of the collection abstract base classes, including Sized, which is used to indicate whether we can determine the number of items in a collection with the built-in len function. Let's import Sized from collections. abc and create a new collection called SizedCollection, which just holds a simple integer size. SizedCollection implements dunder init to initialize the size, and dunder len to retrieve the size via the built-in len function. As you can see, implementing a dunder len method is sufficient to be considered a subclass of the Sized abstract base class. It's worth bearing in mind that our implementations of dunder instancecheck and dunder subclasscheck in SwordMeta are somewhat naïve, as they make no attempt to check the regular, non-virtual based classes of the objects being tested, which could lead to some surprising behavior. Bear in mind that correctly overriding dunder subclasscheck and dunder instancecheck on your own metaclasses is difficult. Don't worry though, we'll be presenting some more digestible alternatives shortly.**

**Non-transitive Subclass Relationships**

**Overriding subclasscheck affords class implementors a great deal of flexibility, so much flexibility, in fact, that not only should you not expect symmetry between the superclass and subclass relationships, you shouldn't expect transitivity of the subclass relationship. What this means is that if C is a subclass of B, and B is a subclass of A it doesn't necessarily follow that C is a subclass of A. One glaring example from Python revolves around the Hashable virtual base class from collections. abc. Let's import Hashable from collections. abc, and test whether object is hashable. Yes it is. Now let's test whether list is an object. Yes, of course it is. Now let's test whether the list is Hashable. No it is not, so even though list is an object, an object is a Hashable, we cannot say that list is a Hashable. This occurs because list, which remember is a mutable collection, disables hashing by removing the special dunder hash method, which would otherwise be inherited from object, and which the Hashable abstract base class checks for through its dunder subclass check implementation. Some further investigation reveals that the list class sets the dunder hash attribute to None. The Hashable dunder subclasscheck implementation checks for this eventuality and uses it to signal None hashability. This example is also interesting because it demonstrates the fact that even the ultimate base class object can be considered a subclass of Hashable, underlining the lack of symmetry between superclass and subclass relationships in Python.**

**Method Resolution with Virtual Base Classes**

**It's worth bearing in mind that unlike regular base classes, virtual base classes don't play a role in method resolution. We'll demonstrate this by adding a thrust method to the Sword virtual base class. Now let's create an instance of broad\_sword. Invoking swipe on broad\_sword works as expected. Attempts to invoke thrust on subclasses raise an attribute error, and checking further, we can see that sword is not present in the method resolution order for BroadSword. For this reason, it is not possible to call virtual base class methods using super, which relies on searching the mro.**

**Library Support for Abstract Base Classes**

**We've pointed out that implementing dunder subclasscheck and dunder instancecheck correctly can be awkward to get right. Fortunately, the standard library provides some support for implementing abstract base classes in the ABC module, the abc metaclass, along with some other useful pieces of infrastructure, including the ABC base class and the abstractmethod decorator. We'll cover each of these in detail now. The ABCMeta metaclass implements reliable dunder subclasscheck and dunder instancecheck methods, along with some other handy capabilities we'll come to shortly. We'll have our Sword class use ABCMeta instead of SwordMeta, which will allow us to dispose of SwordMeta after we've cannibalized it. Now, of course, ABCMeta doesn't know what it means to be a sword, so the test that was previously in SwordMeta dunder subclasscheck needs to be relocated elsewhere. The ABCMeta dunder subclasscheck method calls the special dunder subclasshook method on our actual class to perform the test. In fact, all Python objects have the dunder subclasshook class method, which accepts the potential subclass as its only argument. We can see this by calling it on object where it returns not implemented. The method should return True, False, or NotImplemented. For classes where dunder subclasshook returns NotImplemented the subclass test continues with the usual mechanism of testing the non-virtual base classes. Boolean values definitively indicate whether the argument class is to be considered a subclass of the base or not. Let's try implementing dunder subclasshook for our sword class with pretty much the same definition we use for dunder subclasscheck in SwordMeta previously. This class method, in conjunction with ABCMeta, is sufficient to make our issubclass and isinstance tests work as expected. SamuraiSword is a subclass of Sword, Rifle is not a subclass of Sword, and an instance of broad\_sword is an instance of Sword.**

**Virtual Subclass Registration**

**It's also possible to directly register a class as a virtual subclass with an abstract base class whose metaclass is ABCMeta using the register metamethod. For example, let's create an abstract base class Text, and register the built-in type, str, as a virtual subclass. Notice that the register metamethod returns the class which was registered, a point we'll return to in a moment. Now we can even retrofit base classes to the built-in types albeit virtual bases. Here we demonstrate that the built-in str class is now considered a subclass of our text class defined at the REPL, and string objects are instances of Text. Because the register metamethod returns it's argument we can even use it as a class decorator. Let's register a class, Prose, as a virtual subclass of Text.**

**Combining Subclass Detection and Registration**

**You need to take care when combining virtual subclass registration with the dunder subclasshook technique because the results of dunder subclasshook takes precedence over the subclass registry. Returning true or false from dunder subclasshook is taken as a definite answer. If you also want registration to be accounted for you should return NotImplemented to indicate not sure. To see this in action let's add a lightsaber, which has no sharpen method, to our example. This class won't satisfy the dunder subclasshook test we defined in Sword, but we still want it identified as a virtual subclass of sort, so we've registered it using the decorator form of Sword. register. Even though we've registered LightSaber with Sword the subclass test returns false. To fix this we need to ensure that dunder subclasshook never returns false because doing so causes the dunder subclass check implementation in ABCMeta to skip the check for registered subclasses. Instead, in the case of a negative result we should return NotImplemented. With this change in place we should exploit shortcut evaluation of the logical operators. Subclass detection now works as expected for implicitly detected subclasses, explicitly registered subclasses, and non-subclasses. Do bear in mind, though, that this somewhat contrived example is designed to demonstrate that you should take care with subclass registration. How useful is our virtual base class, Sword, now? The answer is not very because being an instance of Sword is no longer a useful predicate for the object in question, as we can't guarantee the presence of the sharpen method, which was, if you'll excuse the pun, the whole point of the sword.**

**The ABC Convenience Base Class**

**The ABC module contains a class, ABC, which is simply a regular class, which has ABCMeta as its metaclass. This makes it even easier to declare abstract base classes without having to put the metaclass mechanism on show, which may be an advantage when coding for an audience who haven't yet been exposed to the concept of metaclasses. Using ABC our Sword class can be simplified a little.**

**Declaring Abstract Methods**

**Finally, we reach the aspect of Python's abstract base classes, which most immediately springs to mind when we talk about abstract base classes in general, the ability to declare abstract methods. An abstract method is a method which is declared, but which doesn't necessarily have a useful definition. Abstract methods must be overridden in derived concrete classes, and the presence of an abstract method will prevent its host class being instantiated. Abstract methods should be decorated with the abstractmethod decorator, and their abstractness will only be enforced if the metaclass of the host class is ABCMeta. Let's add abstract methods to our Sword abstract base class. First, we need to import abstractmethod from the ABC module. Python syntax requires that we provide a body for each method, so for swipe and parry we raise a NotImplemented error, but for thrust we provide a default behavior of printing, thrusting. We'll take this opportunity to remind you of the distinction between NotImplemented and NotImplementedError. NotImplemented is a value returnable from predicate functions, which are unable to make a determination of true or false. On the other hand, NotImplementedError is an exception type to be raised in place of missing code. We'll also update dunder subclasshook to match. If we now make BroadSword a real subclass of Sword by explicitly adding Sword to the list of base classes we cannot instantiate BroadSword, as it is still too abstract. We get a TypeError, Can't instantiate abstract class BroadSword with abstract methods, parry and thrust. We must implement all abstract methods for the class to be considered concrete. Notice that in thrust we call the implementation provided in the abstract class using super. Now we can successfully instantiate BroadSword and call its methods. Note, however, that the requirement to implement abstract methods doesn't extend to virtual subclasses, only to real subclasses. We can still instantiate SamuraiSword successfully, even though it doesn't implement all of the abstract methods.**

**Combining Method Decorators**

**What if you need to combine abstractmethod with other decorators? The abstractmethod decorator can be combined with the static method, class method, and property decorators, although care must be taken that the abstract method is the innermost decorator. For properties you can independently mark the getters and setters as abstract. Recall that properties are implemented using descriptors. When implementing your own descriptors you'll need to do a little extra work to ensure your descriptor plays nicely with abstractmethod. Let's look now at how to propagate abstractness through descriptors. Any descriptors which are implemented in terms of abstract methods should identify themselves as abstract by exposing a dunder isabstractmethod attribute, which should evaluate to true. Here in MyDataDescriptor, which inherits from the ABC base class, we've declared dunder get, dunder set, and dunder delete as abstract methods. The dunder isabstractmethod attribute can either be implemented as a class attribute of the descriptor class or can itself be implemented as a property, as we have done here. Implementing as a property is useful in cases where abstractness needs to be determined at runtime, as we'll see in an example shortly. First, though, let's see this in action with a very simple example. We will define a class called AbstractBaseClass, which inherits from ABC. Within this class we'll define two properties called abstract\_property and concrete\_property using the appropriate combinations of decorators. By querying the descriptor objects created by the property decorators we can inspect the dunder isabstractmethod attribute of the two properties and see that they have been marked as abstract and none abstract or concrete, as necessary. This happens because the dunder isabstract method flag is inspected by the ABCMeta implementation when the metaclass creates the actual class object, which hosts the property descriptor. So much for the theory, let's see this in practice.**

**Improving @invariant with ABCs**

**We rounded off the previous module of this course on class decorators by building a class decorator for checking class invariants after every method call and property access. This worked fine for both methods and properties with a single application of the decorator, but with chained invariant decorators the checking didn't work as planned for properties, with only the innermost invariant taking effect. Let's recap. Although the innermost lower bound check works when we try to assign a temperature of -300 Celsius through the Celsius setter, violating the upper bound with a temperature of 1\*10 to the 34 Celsius is not detected as we had hoped. The problem here is that our class decorator is detecting specifically property instances with this fragment where we inspect each attribute of the class being decorated and check whether it is an instance of property. Because our property wrappers of type InvariantCheckingPropertyProxy are not detected as instances of property, they are not wrapped a second time, and the invariant specified in the outermost decorator is not enforced. We promised to use abstract base classes to fix this problem, so let's go. We'll introduce a new abstract base class called PropertyDataDescriptor, which inherits from the ABC convenience class, and which contains three abstract methods, which define the data descriptor protocol, dunder get, dunder set, and dunder delete. PropertyDataDescriptor also includes the abstract property, dunder isabstractmethod for correct propagation of abstractness. Note that because dunder isabstractmethod needs to look like an abstract attribute we have implemented it by applying the abstract method and property decorators in that order. Having defined an abstract base class, we now need some subclasses. The first will be a virtual subclass, the built-in property class, which will register with the base class by calling PropertyDataDescriptor. register(property). The second subclass will be a real subclass. We'll modify our existing property proxy, InvariantCheckingPropertyProxy, to inherit from PropertyDataDescriptor, which will also require that we override the dunder isabstractmethod property. In the implementation of dunder isabstractmethod we delegate the attribute of the same name on the referent property, which we are wrapping. Finally, we need to update the search and wrap logic in invariant checking class decorator to use the more general test for instances of PropertyDataDescriptor, rather than the more specific test for just property. With these changes in place, both invariants are enforced on property writes. We'll create a temperature of 42 Kelvin, then attempt to modify the temperature through the Celsius setter with a temperature of -300 Celsius, which is below absolute zero. This now fails as designed, signaling a violation of the temperature not below absolute zero invariant. Then we test the other invariant by assigning a temperature of 1\*10 to the 34 through the Celsius setter. This also fails as designed, signaling a violation of the temperature below absolute hot classing variant. There's a lot going on in this code with decorators, metaclasses, abstract base classes, and descriptors, and it may seem somewhat complicated. All this complexity is well encapsulated in the invariant class decorator, so take a step back, and enjoy the simplicity of the client code in the temperature class.**

**Summary**

**In this module we've explained Python system of abstract base classes, which is somewhat more flexible than similar concepts in other languages. In particular, we covered how the behavior of the built-in issubclass and isinstance functions can be specialized for a base class by defining the dunder subclasscheck and dunder instancecheck methods on the metaclass of that base class. Specialized subclass checks allow us to centralize the definition of what it means to be a subclass by gathering look before you leap protocol checks into one place. Any class which implements the required protocol will become at least a virtual subclass of a virtual base class. The standard library ABC module contains tools for assisting in the definition of abstract base classes. Most important amongst these tools is the ABC metamethod class, which can be used as the metaclass for abstract base classes. Slightly more conveniently, you can simply inherit from the ABC class, which has ABCMeta as its metaclass. ABCMeta provides default implementations of both dunder subclasscheck and dunder instancecheck, which support two means of identifying subclasses, a special dunder subclasshook class method on abstract base classes or a registration method. Dunder subclasshook accepts a candidate subclass as it's only argument, and should return true or not implemented. False should only be returned if it is designed to disable subclass registration. Passing any class, even a built-in class to the register metamethod of an abstract base class will register the argument as a virtual subclass of the base class. An abstractmethod decorator can be used to prevent instantiation of abstract classes, and require methods to be marked as such to be overridden in real, although not virtual, subclasses. The abstractmethod decorator can be combined with other decorators, such as staticmethod, classmethod, and property, but abstractmethod should always be the innermost decorator. Descriptors should propagate abstractness from underlying methods by exposing the dunder isabstractmethod attribute. Python is a large and complicated language with many moving parts. We find it remarkable that much of this complexity is so well hidden for the vast majority of the time in Python. We hope in this course we've given you deeper insight into some important mechanisms in Python which, while a bit trickier to understand, can deliver great expressive power. You've now reached the end of this course in Advanced Python, and now is the time to take what you've learned and apply these powerful techniques in your work and play with Python. If you'd like written material to go along with this course you should check out our Python Craftsman book series. The books correspond to our Pluralsight courses, which cover the core Python language. The first book in the Craftsman series, The Python Apprentice, corresponds to our Python Fundamentals Pluralsight course. The second book, The Python Journeyman, corresponds to our Python Beyond the Basics Pluralsight course. The trilogy is completed by the The Python Master, which corresponds to this Advanced Python course. Pluralsight viewers can follow the URLs shown below each book to get them at a deep discount. If you'd like to dig further into some advanced uses of Python you might care to swing by our Sixty North Blog, Good with Computers, where we cover some very advanced topics, such as integrating Python with C++ code and implementing a sophisticated functional programming technique called transducers in Python. We'll doubtless be back with more content for the ever-growing Python language and library. Please remember, though, that the most important characteristic of Python is that, above all else, it's great fun to write Python software, so enjoy yourselves.**