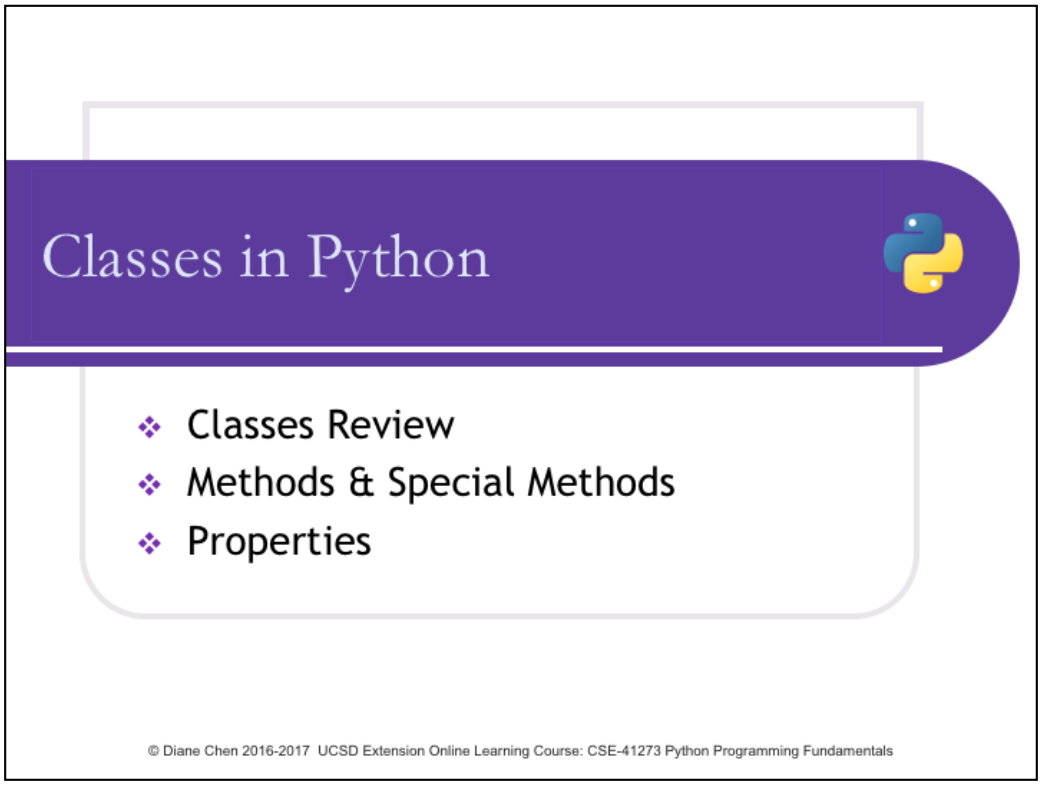
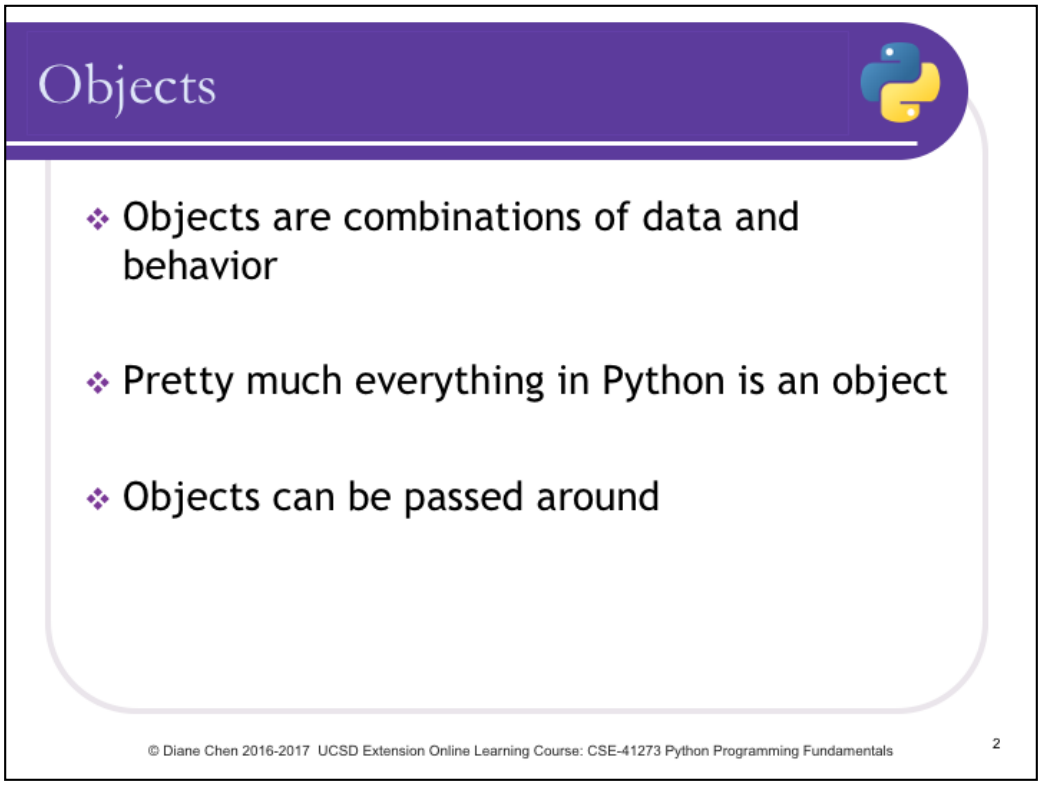
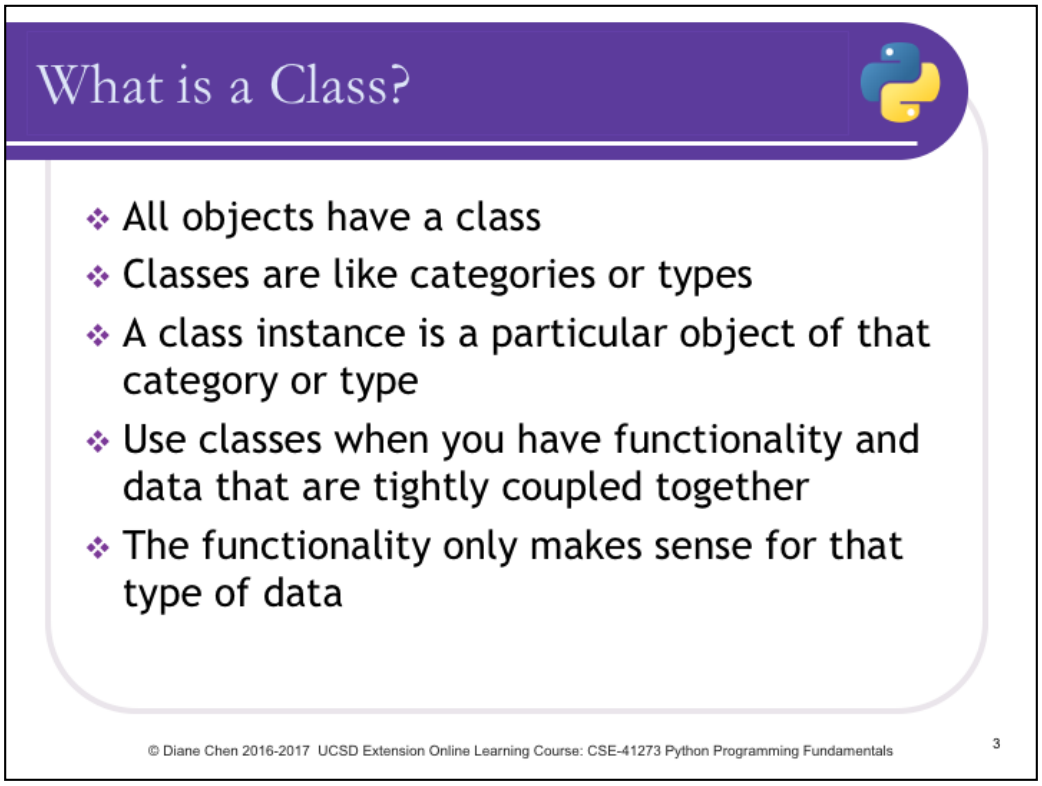


This week we're going to talk about defining our own classes in Python. I'll talk about class methods and some special methods. Later we'll talk about properties and how we can customize behavior of our classes.

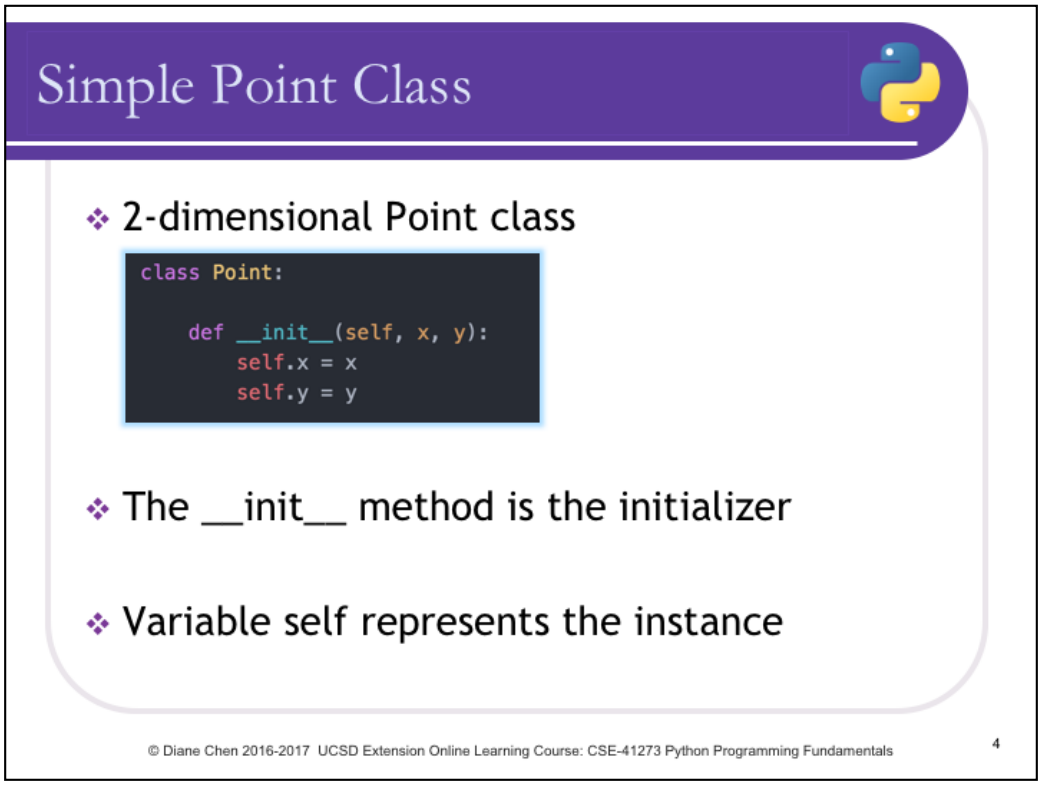


What are objects? As we discussed in week 1, pretty much everything in Python is an object. You can think of an object as a combination of data and behavior. We can assign variables to objects. We can pass objects around, for example as parameters to functions. In fact, functions are objects, too, and function objects can also be passed as parameters to other functions. We'll see some of that later in the course.



All objects in Python have a class. What's a class? Classes represent categories and a class instance is a particular thing in that category. A class is a type of thing. Classes describe attributes and features or functionality of these things.

When would we want to use classes? Classes are useful when you have functionality and data that should be tightly coupled together. Especially when the functionality is unique to that data and wouldn't make sense for other kinds of data. For example let's say we have x-y coordinates we'd like to represent them with a point class. We want to be able to make a point and do stuff with it.

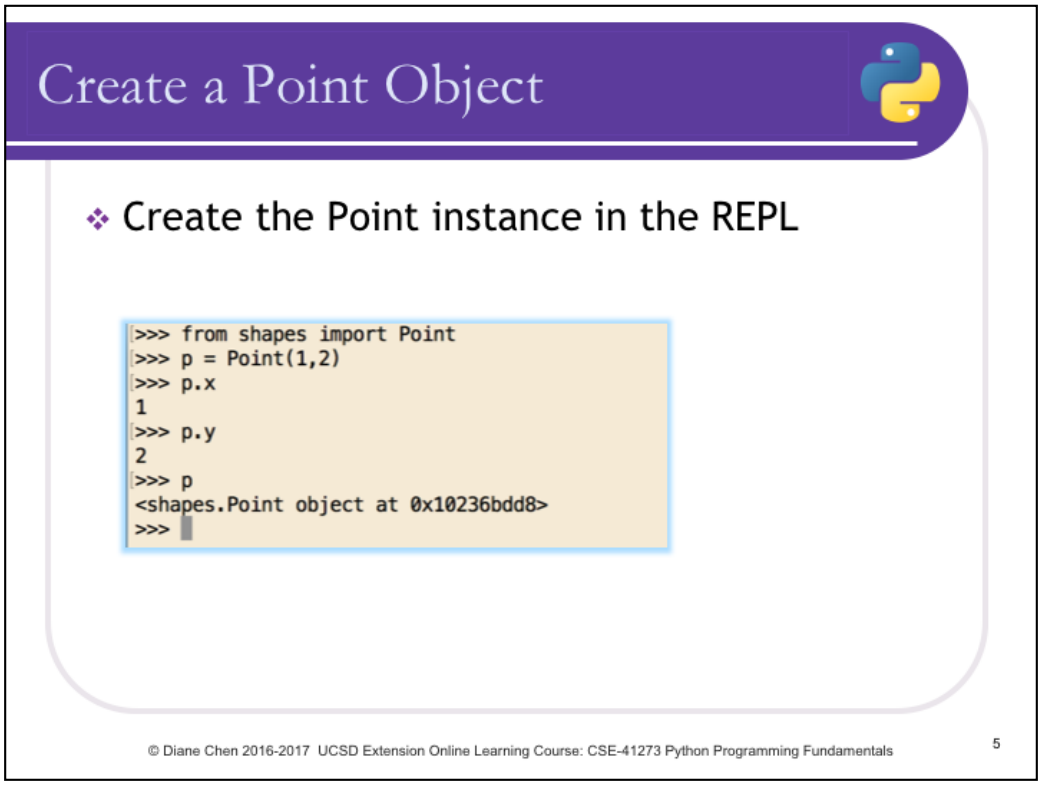


Let's make a simple example of a 2-dimensional point class. This is all we need to start with. We declare the class keyword and the class name, with a colon to indicate to Python that there is more to come. Methods of classes are functions that live within the class definition; as you can see here, it is in the indented block following the class statement.

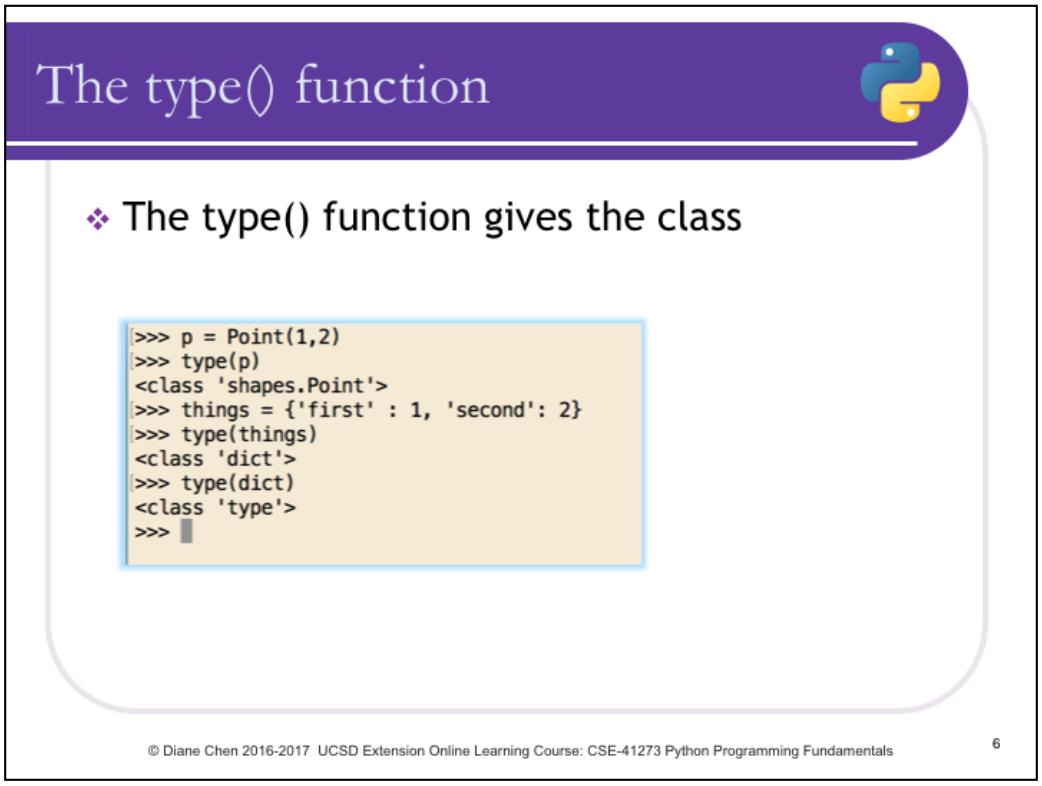
The method "dunder init" is the initializer method. You may hear some people refer to it as a constructor method. That's not completely true, since the object has already been created when we get to this point, and here we initialize whatever needs initializing on the already-created instance object.

The variable "self" refers to the instance that we just created. You don't have to call it "self", but there is very, very strong convention for using that name, because the first parameter of an instance method always is a reference to the instance object.

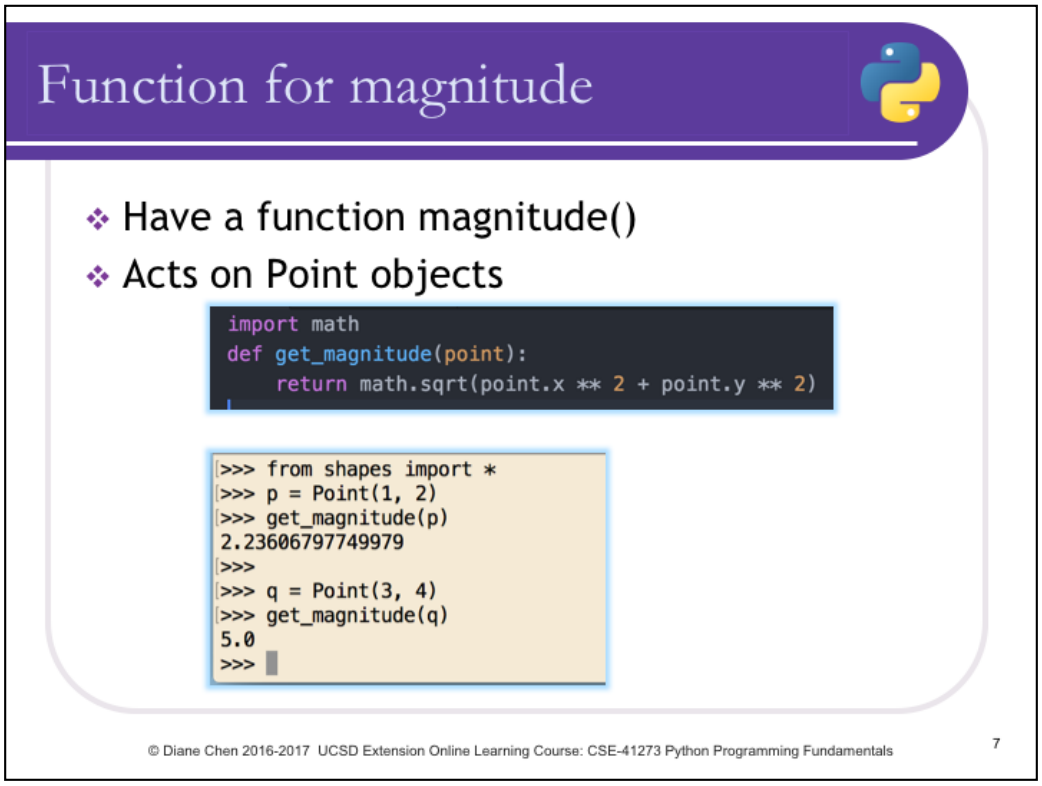
We have 2 variables x and y as input to the instance, so we add the attributes x and y to the instance "self" and assign them the input values.



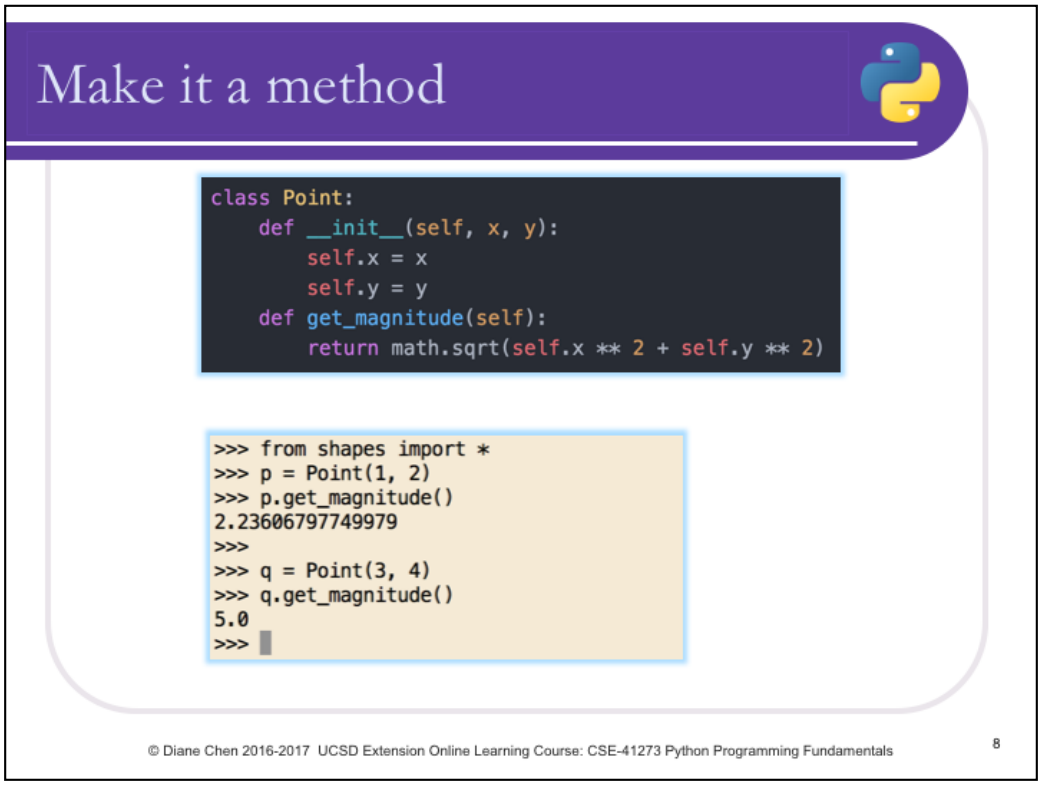
Put the code for the point class in a file called shapes.py. Then we can test it in the REPL. We import the Point class, and create a point p that is an instance of Point. We can print the attributes x and y of our point. When we just print p, we see that it is a shapes.Point object. That information isn't really very helpful because it doesn't have any detailed information about the instance itself, other than its type. But don't worry, we'll fix that later.



We've already seen the built-in function, type(), that will display the class if we need to know it. Type() can be handy when you're in the REPL if you're not sure what object you have. Since everything has a class, we can use the type function on anything, including a class.

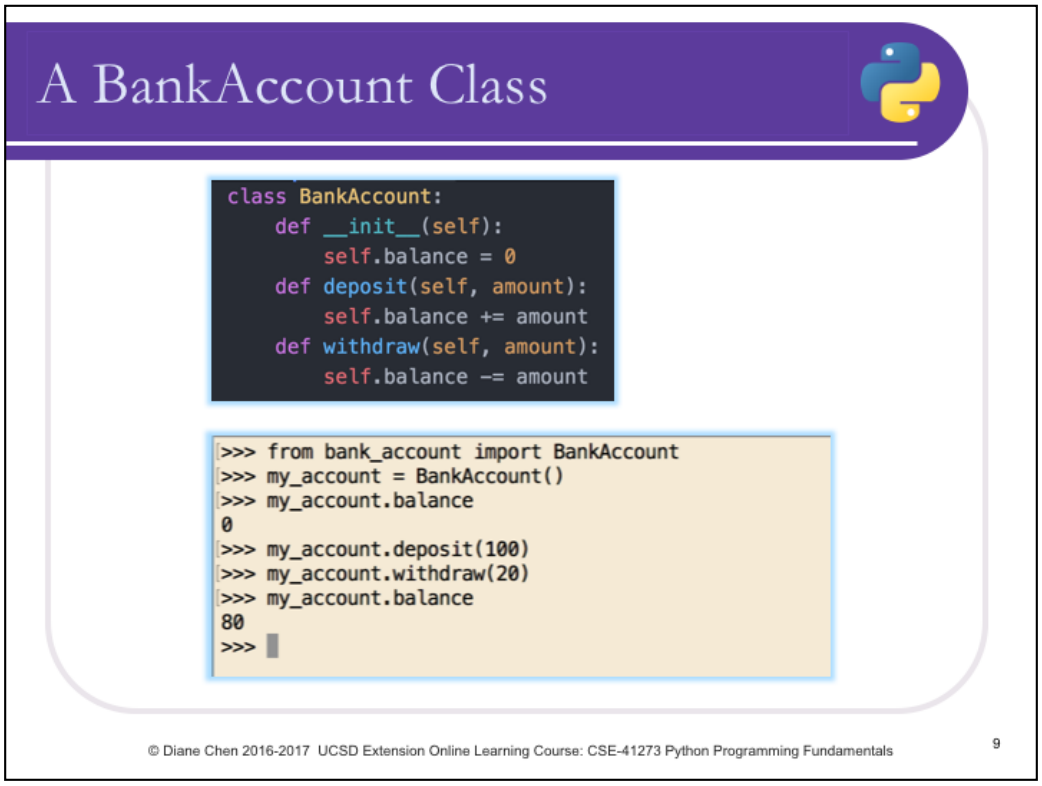


Let's add a function to shapes.py that calculates the magnitude of a point; if we think of the Point as a vector from (0,0) to the point coordinates. It will act on a given Point object. Let's see how we use it. Remember we have to exit and restart the Python REPL whenever we make changes to an imported file. In a fresh REPL, let's import everything from shapes. Normally that is not good practice because it can pollute the namespace with unnecessary things, but it makes life easier for us now, and we're just testing in teh REPL anyway. We create a point and run get_magnitude on it to get the magnitude. That's pretty easy. But the get_magnitude function only applies to Point objects; if we gave it a different kind of object, we would get an error. So it only makes sense to make it a method on the point object.



Here we see I've moved the get_magnitude function to be a method in the point object. Note I've sacrificed the usual blank lines in the code to save room on the slides. Don't do that in your homework!

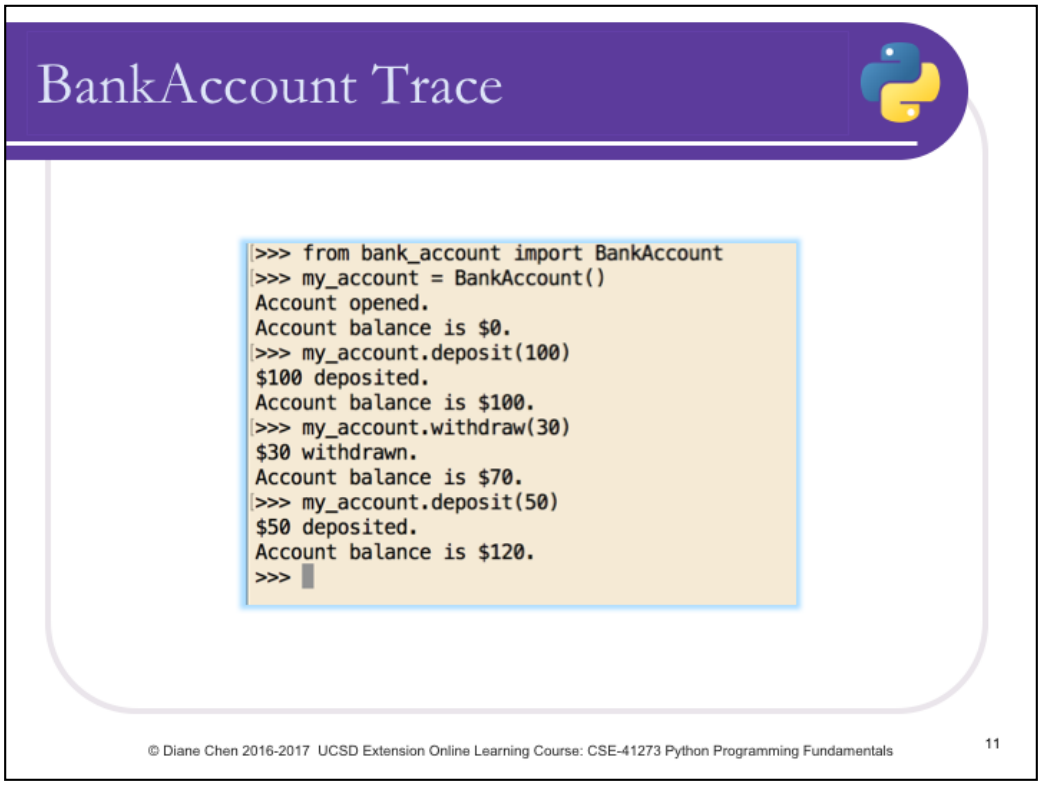
I changed the parameter to follow the naming convention of "self". Now, in a fresh REPL, we can apply the method get_magnitude() to the point objects. So now the point objects know how to get their own magnitude. Having objects manage themselves in this manner is a concept of being object-oriented.



We'll get back to the Point class again later. Here is another example of a class. Let's start making some software for banking. We create a BankAccount class. Note that the class name follows the CapWords or Capital CamelCase convention. This is the preferred naming convention for classes. You will find many that do not follow this convention, but if you are making new classes, there is no reason not follow the accepted naming conventions. Again, for the sake of space on these slides, I'm skipping docstrings and the normal blank line spacing. We can create bank accounts and deposit or withdraw from them. Since the balance is an attribute, we can print the balance in the REPL.

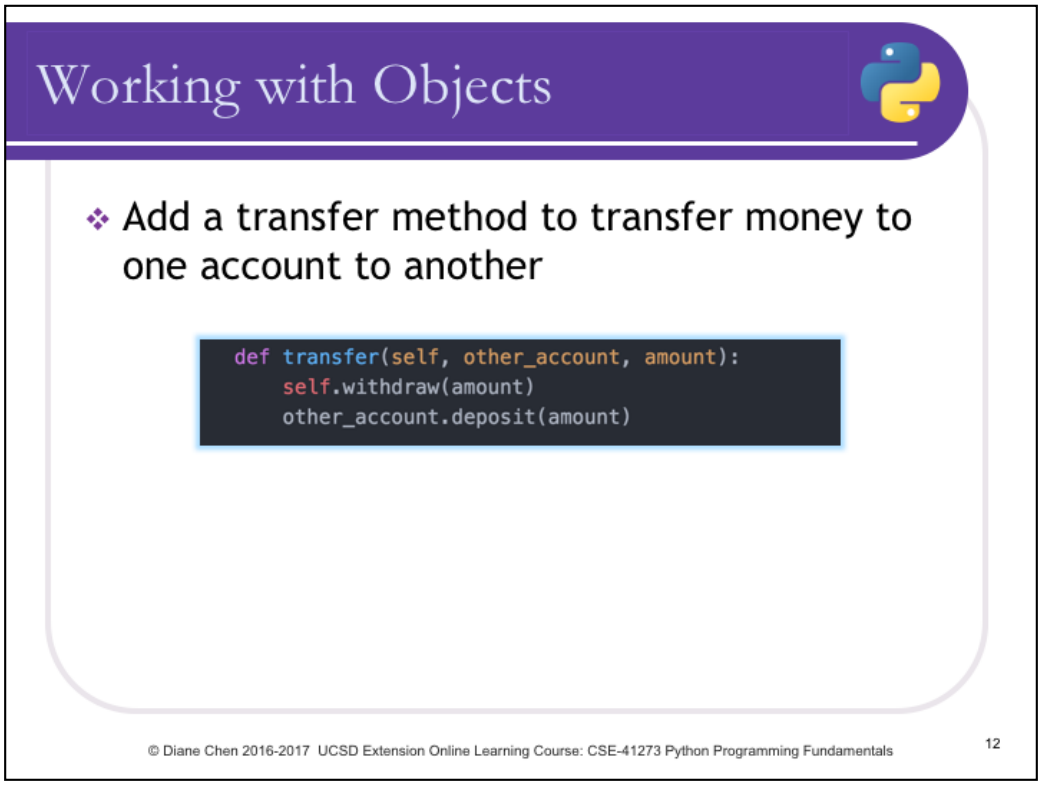
Now let's add some print statements so we can see what's going on. A print_balance method also helps us keep track of how it is working and the effect on the account. Each method call prints information about the operation being performed, then prints the balance.

BankAccount Trace



```
>>> from bank_account import BankAccount
>>> my_account = BankAccount()
Account opened.
Account balance is $0.
>>> my_account.deposit(100)
$100 deposited.
Account balance is $100.
>>> my_account.withdraw(30)
$30 withdrawn.
Account balance is $70.
>>> my_account.deposit(50)
$50 deposited.
Account balance is $120.
>>>
```


Here are the results of creating a bank account and doing some operations on it. First we open an account and it prints that the account is opened and the balance is zero dollars. Then we deposit $100, and it prints out that $100 was deposited and the balance is now $100. Then we withdraw $30, it prints that information and the balance is now $70. Now let's deposit $50. It prints that and the balance is now $120. So that gives us a good feel for what is going on when we are operating on the bank account.

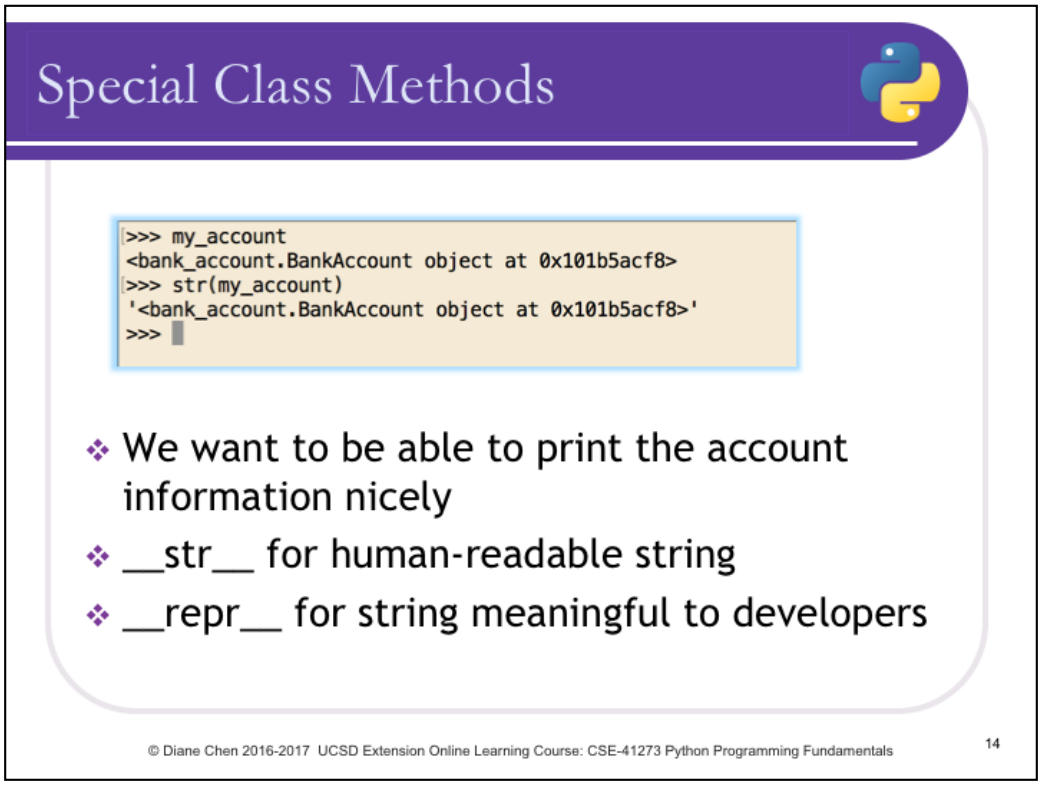


To show an example of working with objects, let's add a transfer method to the bank account so we can transfer money from one account to another. The transfer method has 2 arguments. The first one is another BankAcount object and the second one is the amount to transfer from this account to the other account. Note that the "self" argument is not considered an input argument because it is added automatically by the system.

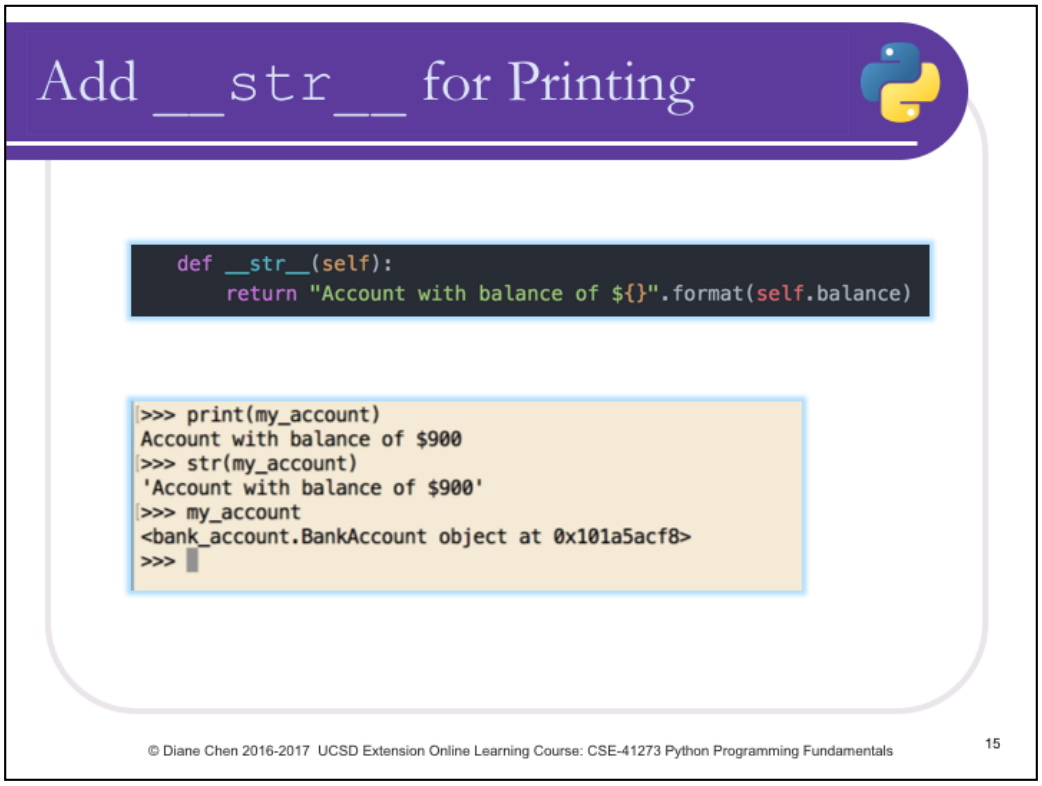
Transfer Money

```
>>> from bank_account import BankAccount
>>> my_account = BankAccount()
Account opened.
Account balance is $0.
>>> my_account.deposit(1000)
$1000 deposited.
Account balance is $1000.
>>> sue_account = BankAccount()
Account opened.
Account balance is $0.
>>> sue_account.deposit(350)
$350 deposited.
Account balance is $350.
>>> my_account.transfer(sue_account, 100)
$100 withdrawn.
Account balance is $900.
$100 deposited.
Account balance is $450.
>>>
```

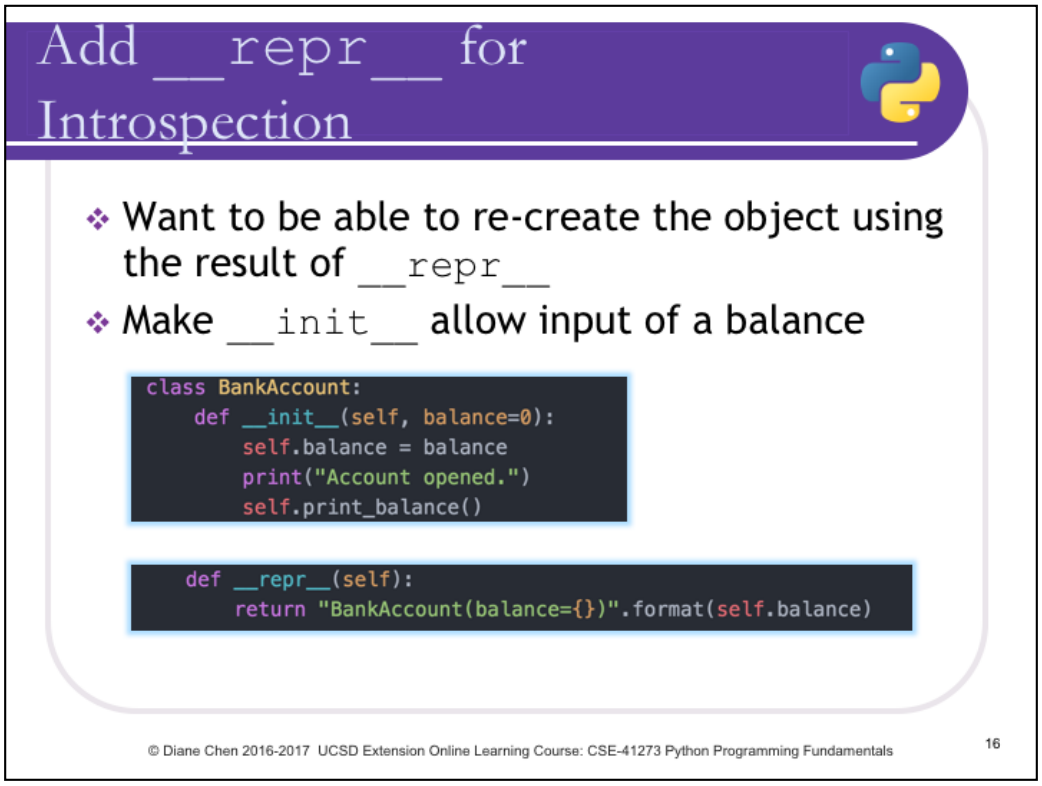

Here is a trace of what is happening when we create 2 bank accounts and transfer money. First we open my_account and add $1000. Then we open sue account and depost $350. Then we transfer from my_account into sue's account, $100. We can see that $100 is withdrawn and then $100 is deposited. Of course we can't tell which account is being withdrawn or deposited. But at least we see a trace of what's going on. Later we'll add names to the bank accounts so that we can keep track of which one is which in our logs.



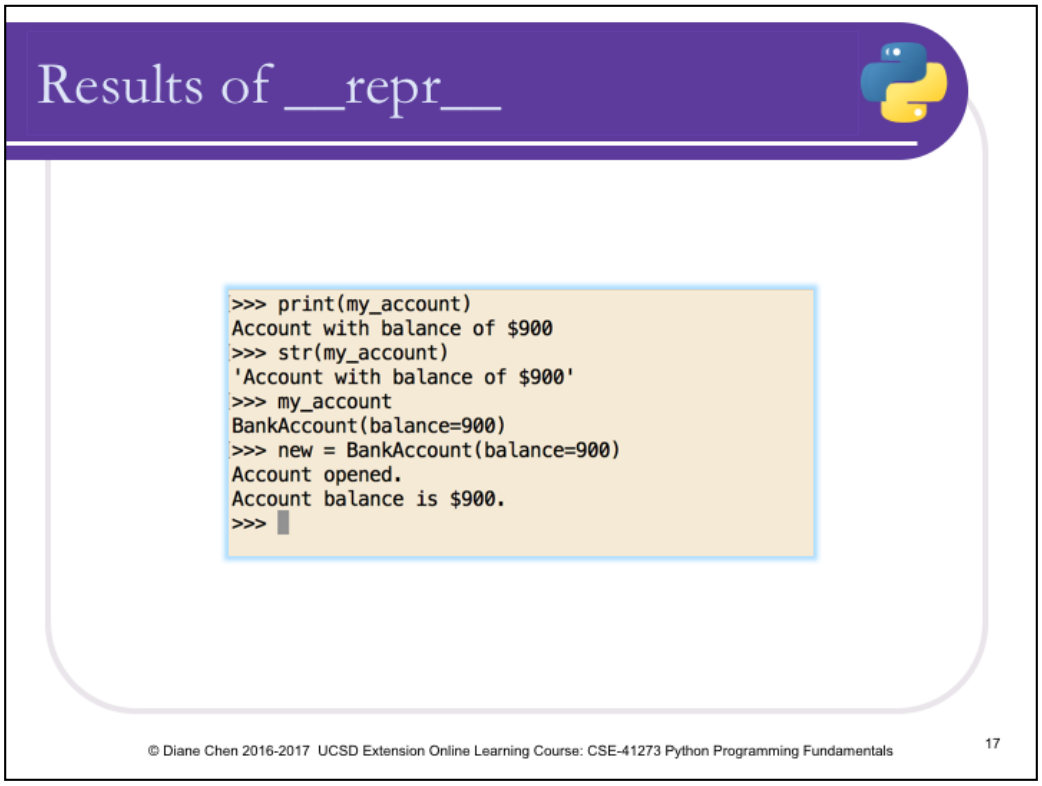
What if we want to print information about an account? When we print it in the REPL, we only get the class information - not very useful. Even if we use the str() function, we just get the same string back. The str() function is used by the print() function. So if we wanted to show bank account information nicely when we use the print function, we need to implement str(). How can we do that? There are two special dunder methods that we can implement to get information about the object. Dunder str() is for a human-readable, nicely formatted string; something that you would want to see from the print function. Dunder repr() is for a string that is meaningful to developers.



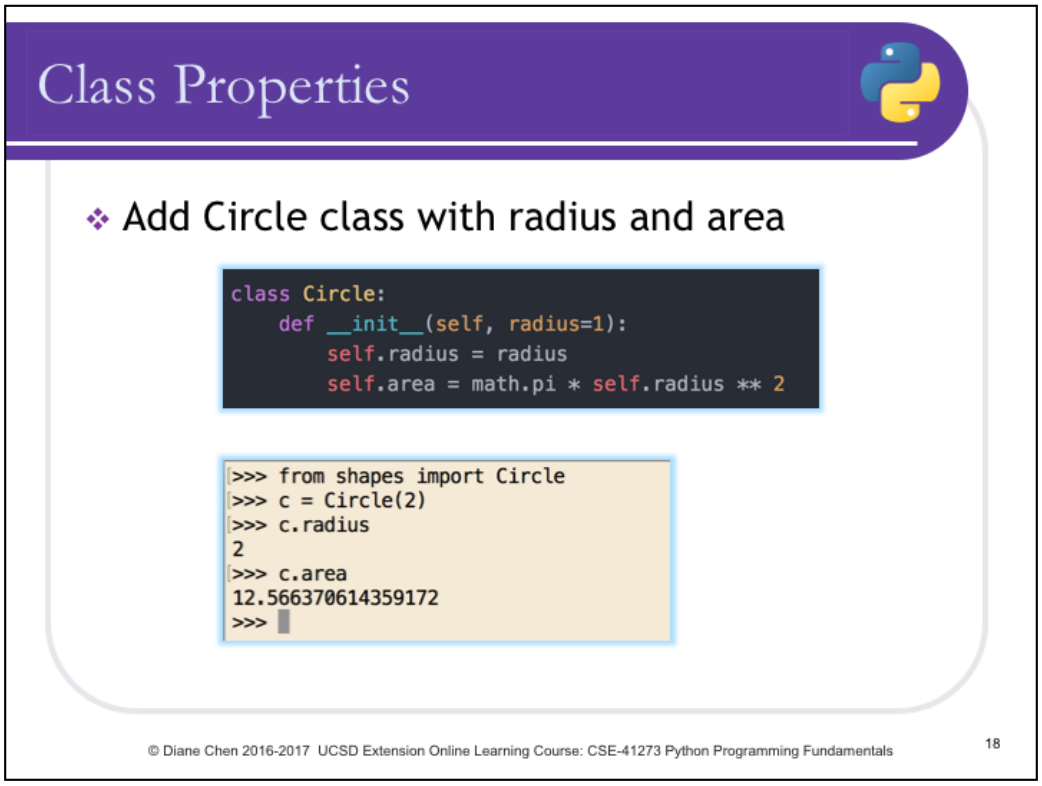
Let's implement the dunder str function first. Here we print out a nice string of information. When we print my_account, it prints "Account with balance of $900". This is nice! But when we just print it on the REPL, we get the same string as before.



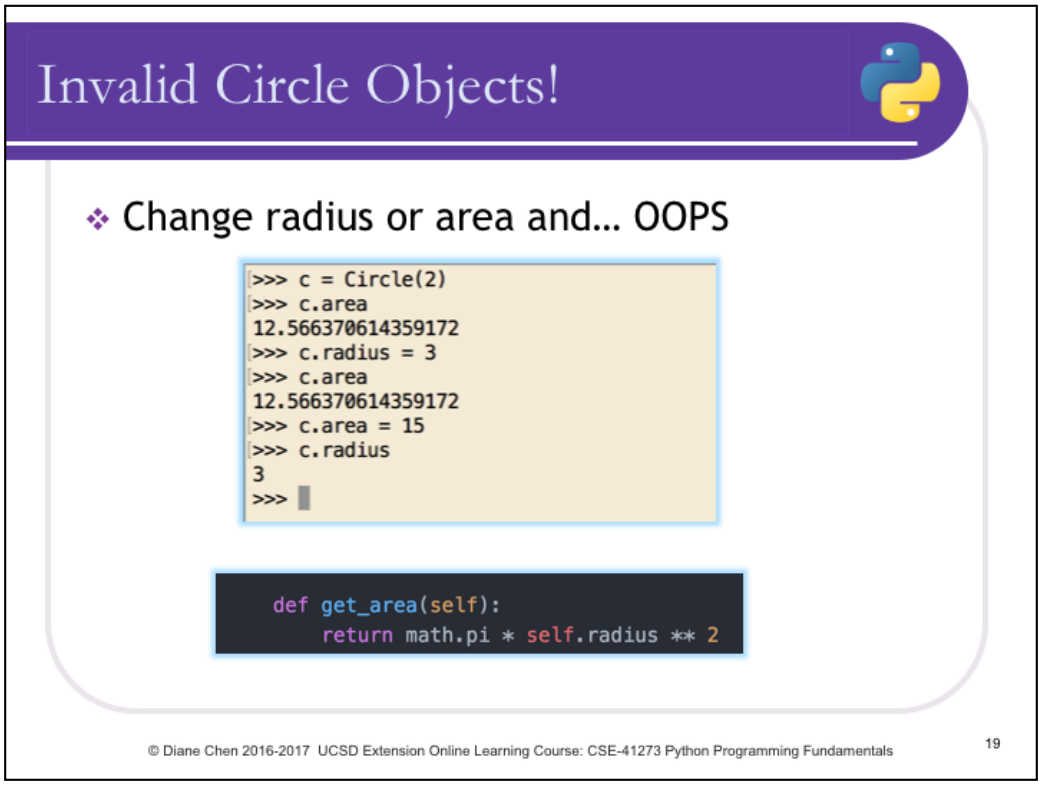
One of the "conventions" of using dunder repr is that we should be able to re-create the object with the information given. In order to do that, we need to be able to create an account with a specified balance. We're going to change the dunder init method to allow an optional balance that defaults to $0. Then in our dunder repr function, we print out the class name of bankAccount with balance information.



We start up the REPL again and input all the account info as before - I haven't shown that here. Now we can see the print gives us a nice string. When we print it from the REPL, it prints out something that we can copy and paste to make a new account containing the same information as the original. This is really helpful during debugging in the REPL. In a program, if we were writing information to a log file, using the dunder repr function when we print to the log file can be very helpful for tracking and debugging.



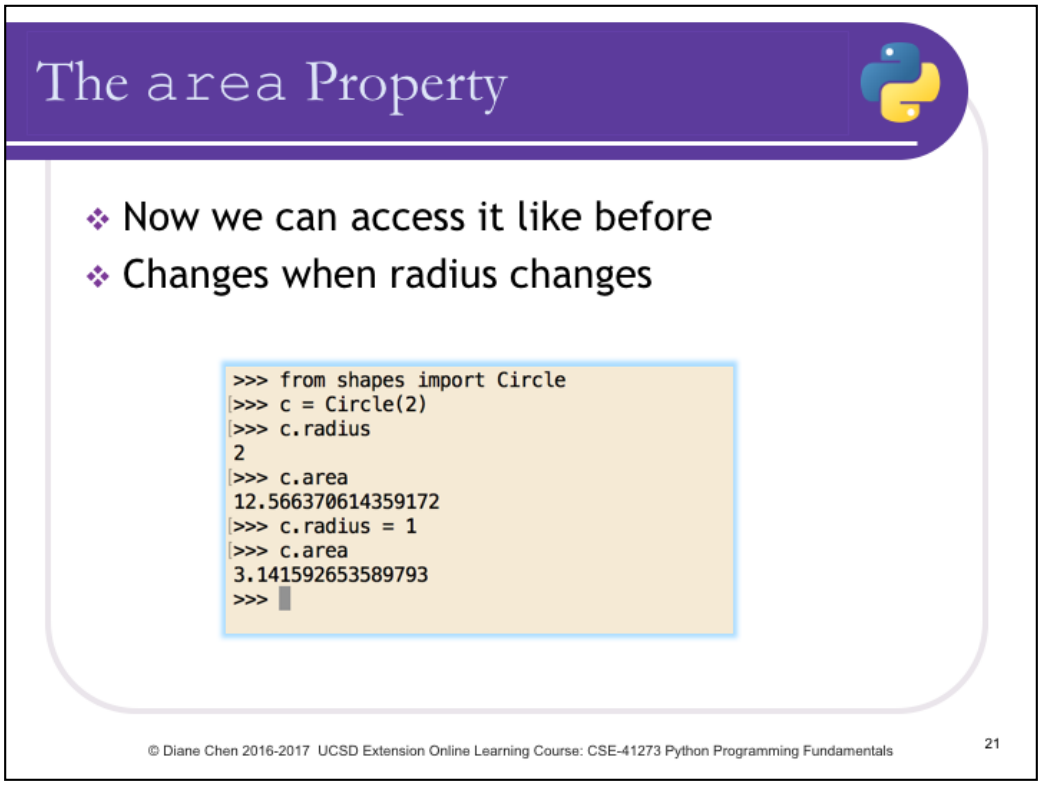
Now let's talk about class Properties and how to use them. Let's add a Circle class to shapes.py and give it properties of radius and area. Looks good, right? Try it out! We create a circle c with radius of 2, and the area is calculated for us. Seems pretty cool, but can you see where we might have a problem?



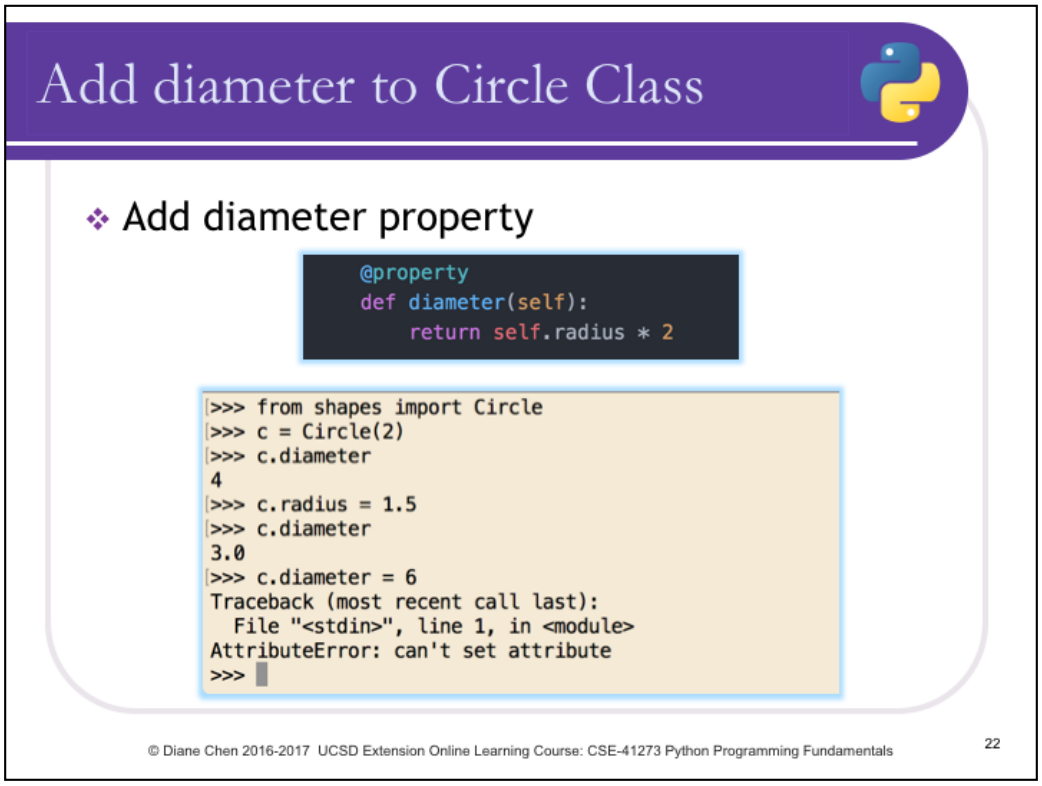
If we change the radius of the circle, the area doesn't change. Similarly, if we change the area, the radius remains the same. This is not good at all! Now we have invalid Circle objects. What can we do to prevent this? We could remove the area property and make a "getter" method for the area as shown here. This works, but it is awkward – now we have to type "c.get_area()" with parentheses instead of just "c.area" – and it's starting to look too much like Java code! Let's not do that, OK?

Of course, Python has a solution for that! We create a method called “area” and make it a property using the property decorator. Decorators are a special thing in Python; for now, it’s just important to know that ”@property” is a special modifier for the method that makes the method behave just like a property or attribute.

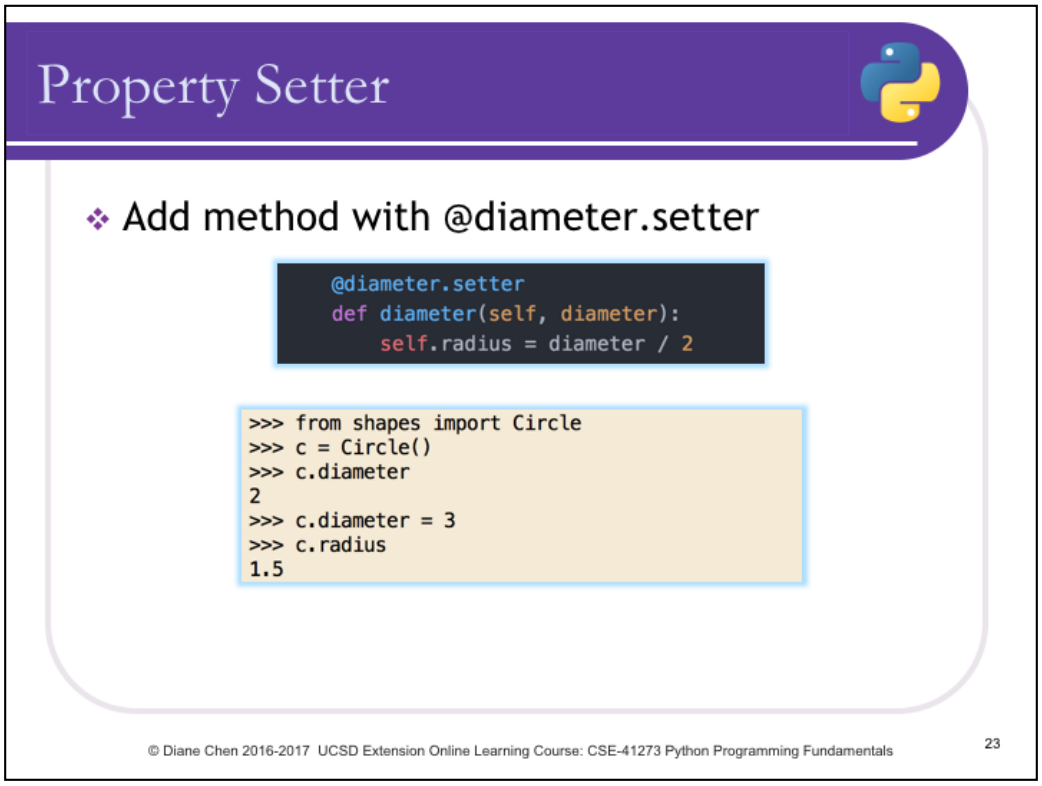
Remember the Point class we created earlier with the get_magnitude() method? We would want to make it a “magnitude” property too.



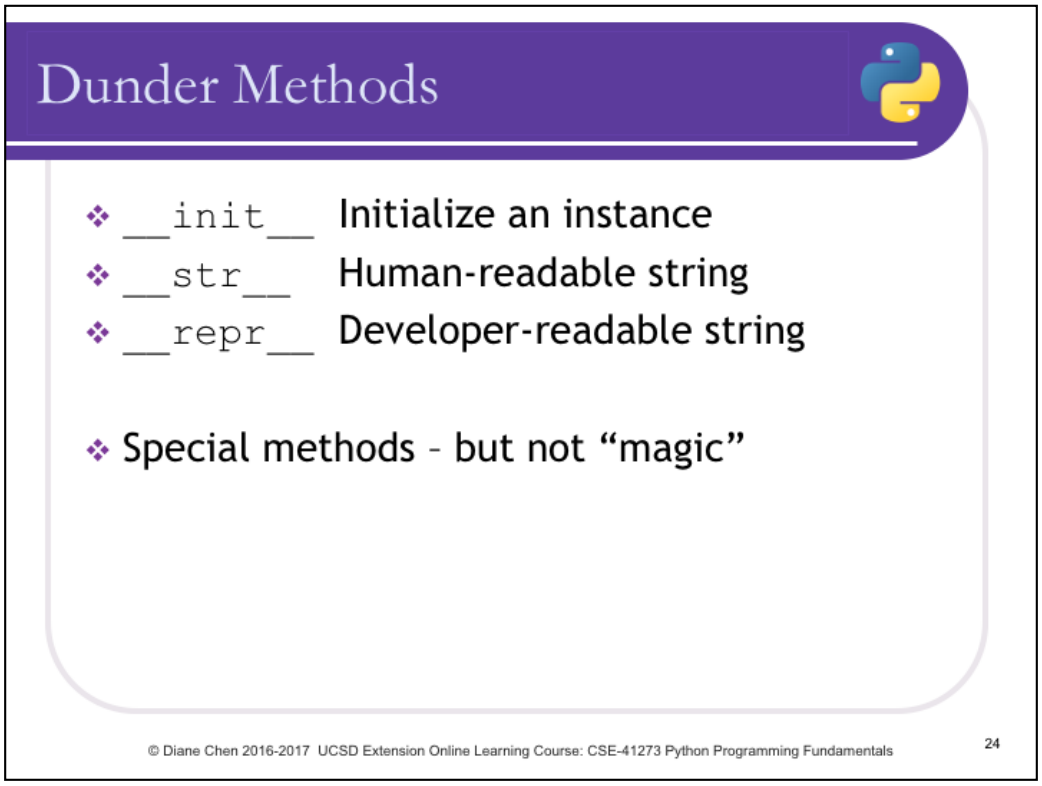
Starting a new REPL, we can see that this is a much better solution. We can access the area just like a regular property. When we change the radius, the area changes to reflect the change. We can't change the area ourselves, but we're not sure we need to do that anyway. Does it make sense to change the area of a circle?



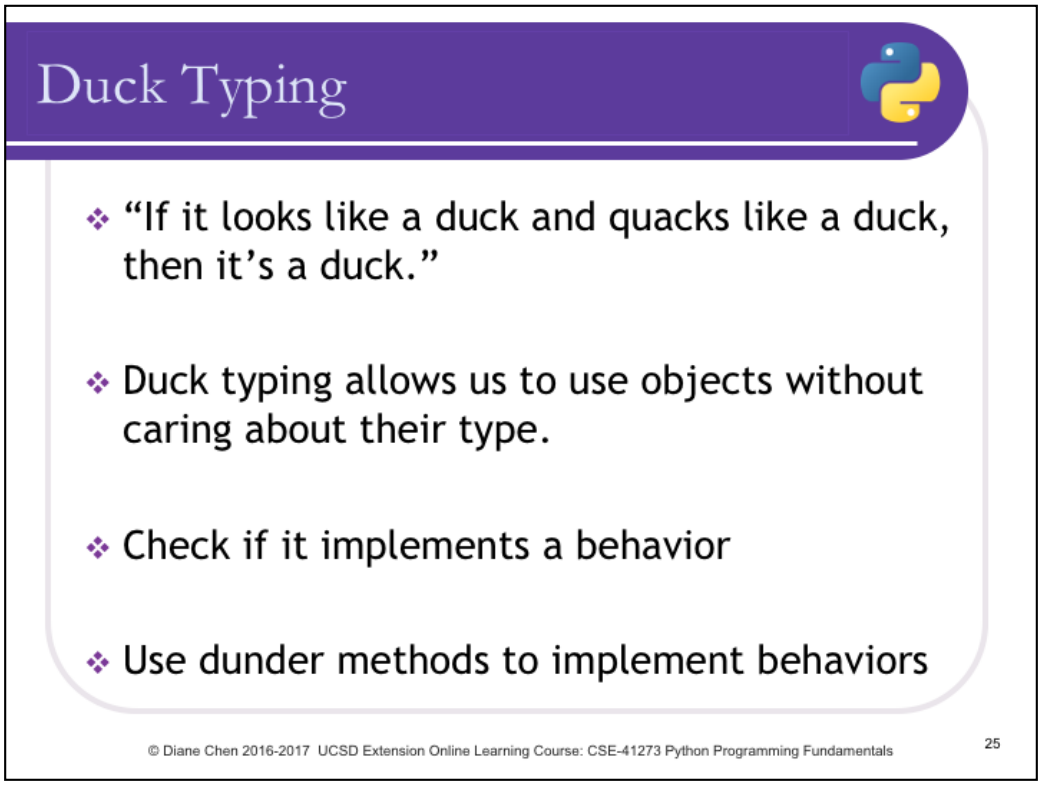
Now that we know about property decorators, let's add a diameter property to the Circle class for convenience. Try it out in the REPL. Don't forget to always restart the REPL whenever there is a change to the program file you are importing. Also don't forget to save the file in the editor so that your changes are actually reflected in the new REPL when you import the file. That is the Voice of Experience talking. It's these kinds of little things that can be annoying and even though minor, the distraction slows down development. So here we see that our diameter updates correctly when we change the radius. However, if we want to change the diameter, we can't because it is a read-only property.



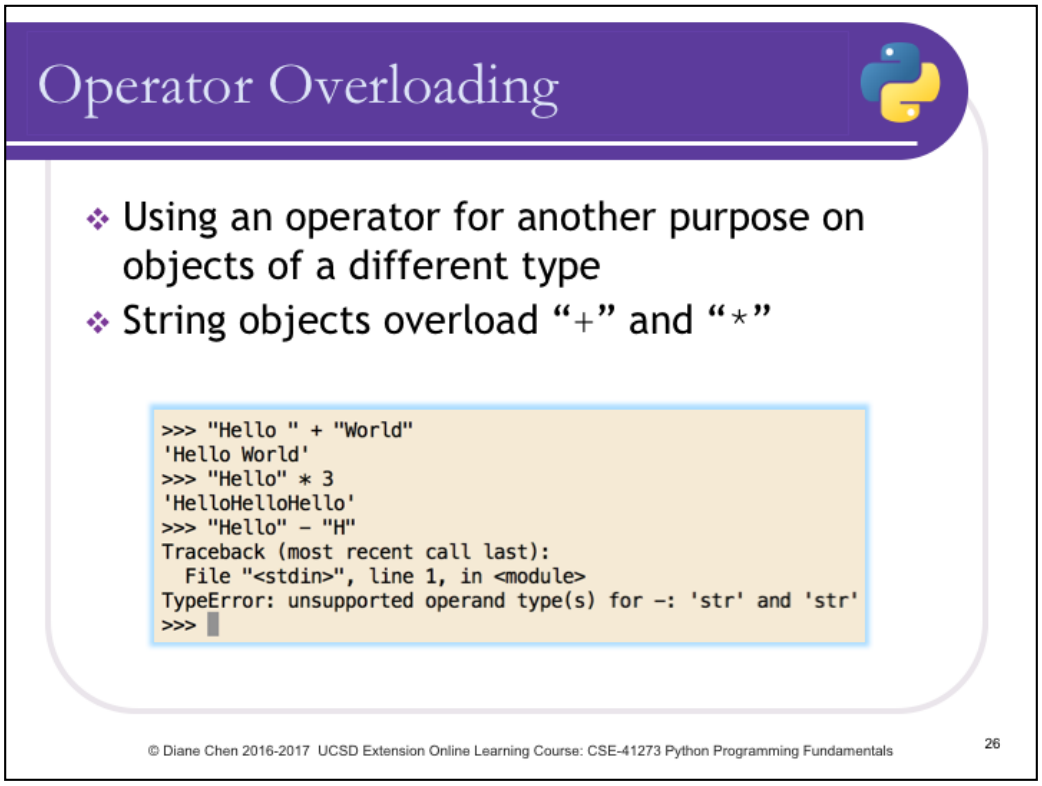
We want to make the diameter into a property that we can change. Can we do that? Of course we can! We simply add another method with a special “setter” decorator applied to it to make it behave just like a property that we are allowed to set. Note that the method name is the same as the first method we made into a property, but it has an input parameter. The decorator syntax may seem a little odd but that’s how it works. Now we are able to change the diameter and the change is reflected in the radius. Pretty cool!



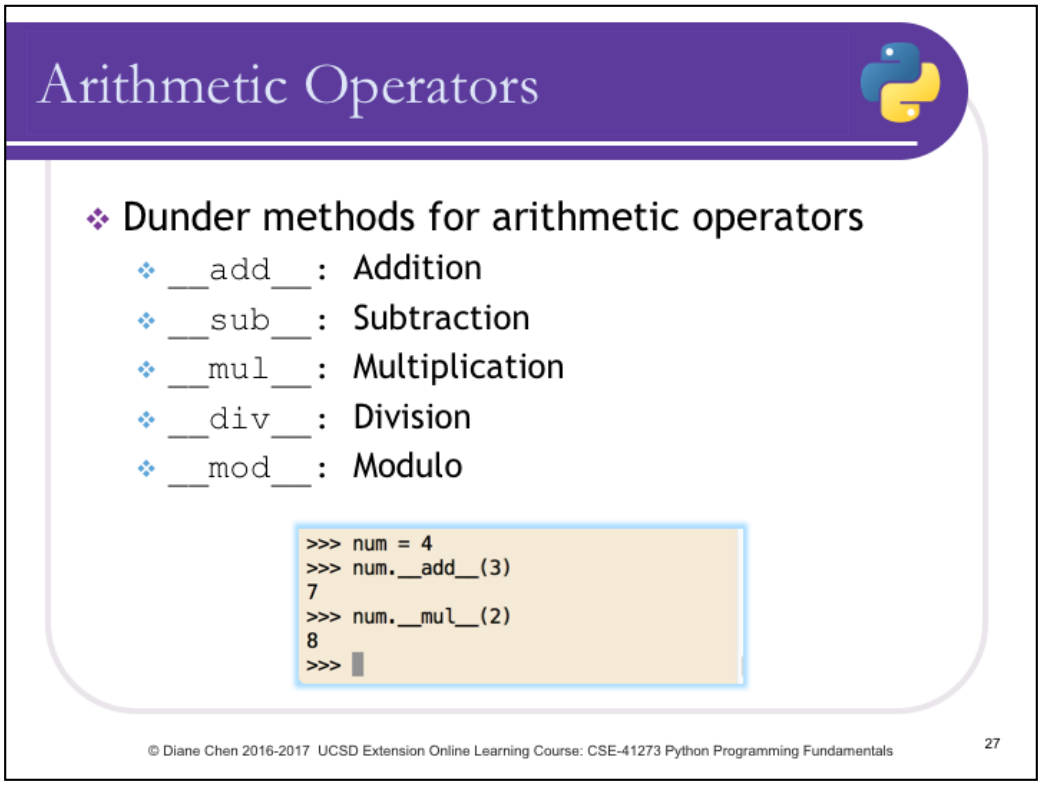
Let's review the dunder methods we've seen so far: dunder init, for initialization of an instance object, dunder str to get a nice human-readable string of the object, and dunder repr to get a string in a format useful for developers. There are other dunder methods for classes. You will sometimes hear or see them referred to as "magic" methods. There is nothing magic about them, they are very straightforward and easy to learn about and use. In my mind, decorators are more "magical" but even they aren't hard to understand – though they may seem magical at the moment.



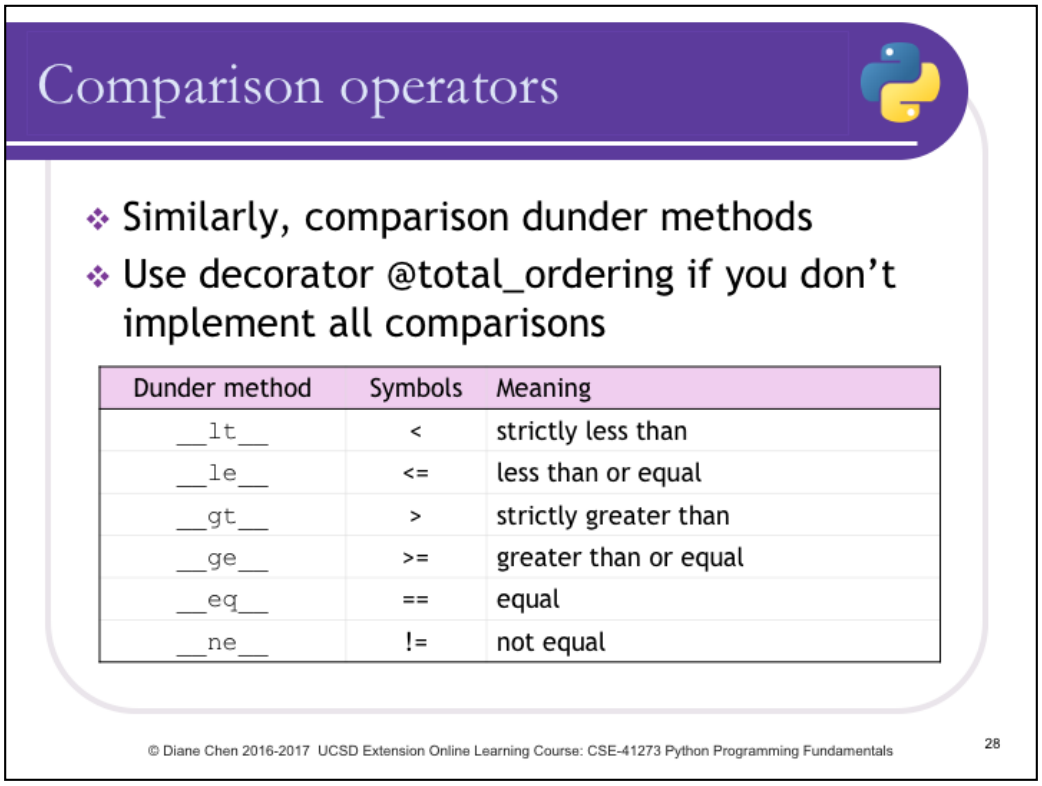
You may have heard the term "Duck Typing". What is Duck Typing? The term comes from the expression “If it looks like a duck and quacks like a duck, then it’s a duck.” Duck typing is an important feature of Python and means that we pretty much never care about the types of objects. Instead of checking what the type of an object is, we instead check whether it implements a feature. If the feature or function we are concerned with is implemented, we don’t care what kind of object we are given. It is sometimes said that Python is a language for consenting adults. We don’t spend time doing things like checking types of objects passed into our code. Dunder methods provide a way for our class to customize operators and other built-in Python behavior for our objects. We use dunder methods to implement specific behaviors that we want our classes to use.



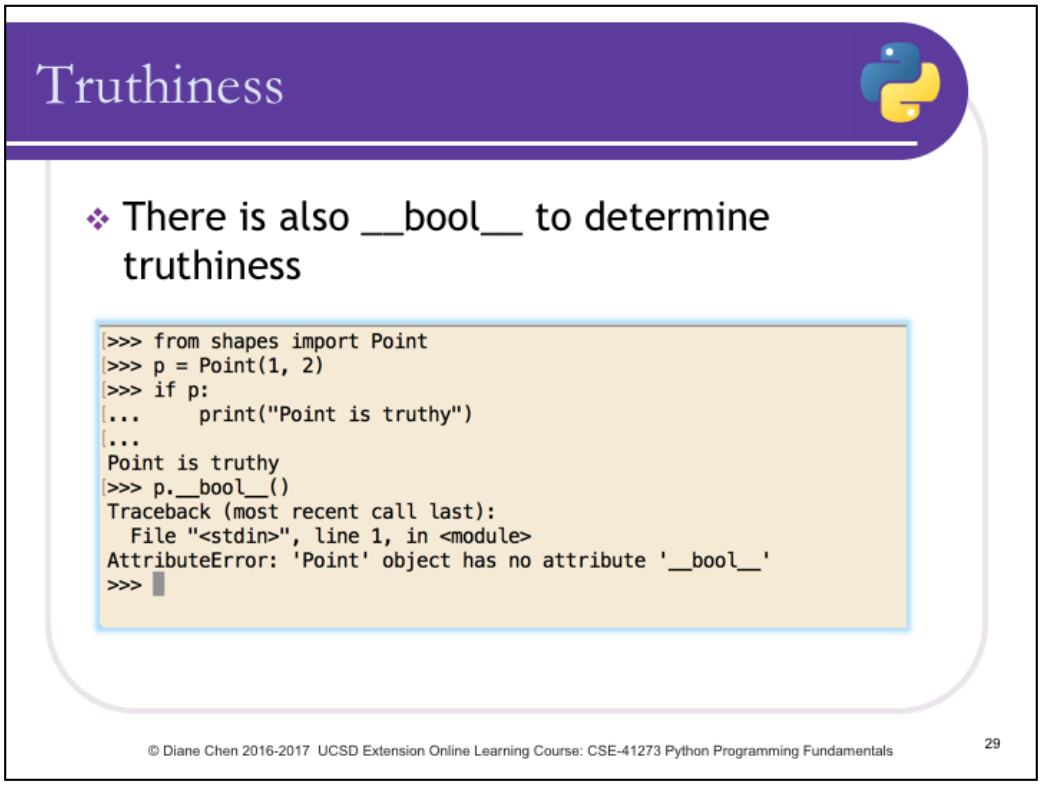
We talked a bit about the term "operator overloading" when we covered lists and strings. What is it really? Let's review a minute. Operator overloading is when we take an operator whose standard meaning is one thing, and we use it to mean something else, usually on an object class for whom the original meaning would not apply. Take for example the arithmetic operators. They have a standard meaning for numbers. But string objects have overloaded the add and multiply operators so that they do meaningful things for strings. You can use the plus sign to "add" two strings together, namely, concatenate the strings. You can also "multiply" a string by an integer to repeat the string that number of times. But subtraction is not supported for strings. It is not overloaded.



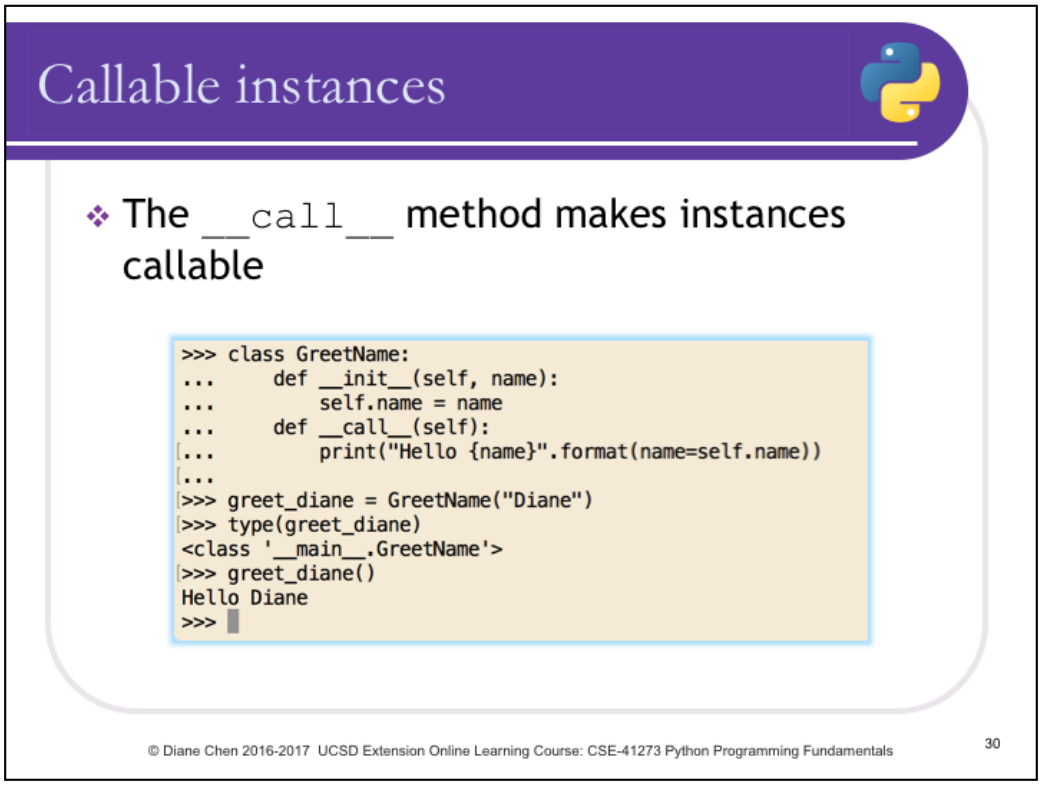
Operators are implemented with dunder methods. Here we see the dunder methods that go with the standard 5 arithmetic operators. Not all types implement all the operators. For example, in Python 3, if you use dunder div on an integer instead of a slash, you get an AttributeError, because division of integers in Python 3 always results in a float. In Python 2, this was not the case. In Python 3, if you want an integer back from integer division, you have to use two slashes, known as the “floor division” operator. It is possible to implement these dunder methods on any object and make them do whatever you want.



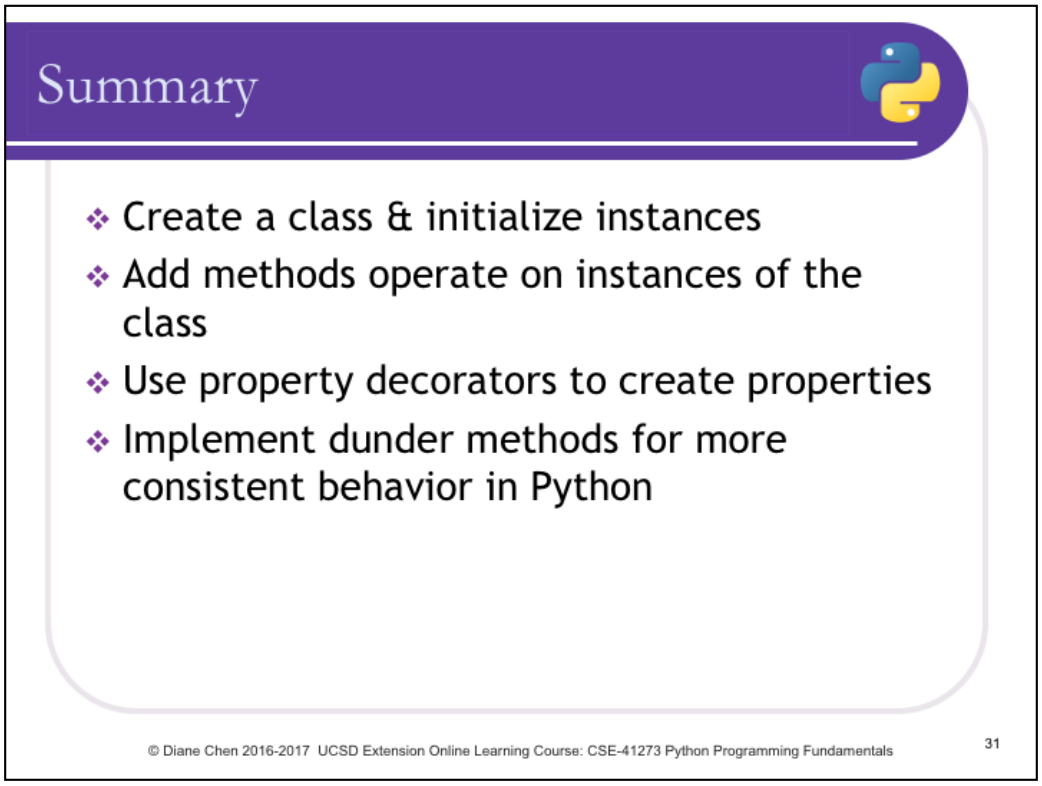
Comparison operators can also be implemented, and here are the dunder methods that make up the comparison operators. It looks like a lot of work with so many methods to implement if you want to be able to compare your class instances. But in reality, you only need to implement 2 complementary operators and Python figures out the rest. For example, if you implement the dunder equal and dunder less than operators, the system can figure out the answers for the other comparisons from those two. There is a decorator “total_ordering” from the functools library, that is applied to the class to tell Python to figure out the operators we didn’t specifically implement. It is a little bit slower to do it this way. If your instances need to be really fast, you might find this to be a bit of a bottleneck. Then you would want to implement all the comparison dunder methods. However, in almost all cases, using the "total ordering" decorator is sufficient.



In addition to the comparison operators there is also dunder bool that is used to determine the truthiness of an object. If dunder bool is implemented, then you can reliably write code with the object in an if statement. I say reliably, because if you don't implement dunder bool on your class, the instances will always be truthy. For example, here we have a Point object that Python treats as truthy, even though it has no dunder bool method, so any testing of truthiness of the instance is meaningless. Clearly, you don't always need to implement it – in any case, I'm not sure how you would implement dunder bool on a Point. How would you decide when a point is falsey? After all, a point at (0,0) is still a legitimate Point. But sometimes you might have a class where you want to be able to say "if instance" in your code; then you would need to implement dunder bool.



There is also a dunder call method that allows instances of the class to be callable. You can think of it as making the instance object so it can be used as a function. Here we define a class GreetName with a dunder call method. When the instance is created, it is given a name that is stored in the instance. Then when we call the instance like a function, the dunder call method executes and prints ”Hello” with the name stored in the instance.



So, let's review what have we learned this week. We know how to create a class and initialize objects of the class, called instances. We can add methods to the class to do things with our instances. We can add methods and decorate them with @property decorators to make them operate like properties or attributes. We learned how to implement some dunder methods so our new class instances will behave nicely in Python and are easier to use.

In the resource notes for this week, there are some links to more information about classes and creating them. Next week we will learn about inheritance and some other aspects of creating our own classes.