

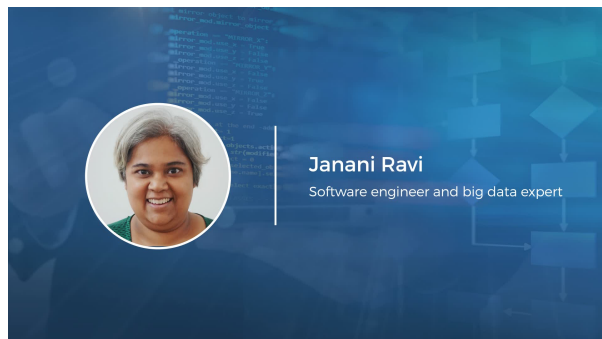
Python Classes & Inheritance: Advanced Functionality Using Python Classes

Examine the advanced features that you can implement by using classes in Python, such as special methods, iterators, class properties, and abstract base classes. Key concepts covered in this 14-video course include how to represent objects by using customized strings; performing addition operations on custom objects; and performing subtraction operations on custom objects. Next, observe how to perform multiplication operations on custom objects and perform floor division, modulo, and power-of operations. Then study learn built-in functions to work with custom data types; learn to execute for-loops on custom data types; and learn about properties on classes for intuitive use. Learn about properties using a simpler syntax; work with class methods to access and update class state; work with utility methods on classes; and learn how to use the abstract method to make classes that are not instantiable base classes. Finally, learners will be shown special methods and what they represent; learn to define a class and create a property within it; and observe how to differentiate between class methods and static methods.

Table of Contents

- [1. Video: Course Overview \(it_pycipydj_04_enus_01\)](#)
 - [2. Video: The repr and str Special Methods \(it_pycipydj_04_enus_02\)](#)
 - [3. Video: The add Special Method \(it_pycipydj_04_enus_03\)](#)
 - [4. Video: The sub Special Method \(it_pycipydj_04_enus_04\)](#)
 - [5. Video: The mul Special Method \(it_pycipydj_04_enus_05\)](#)
 - [6. Video: Special Methods for Other Operations \(it_pycipydj_04_enus_06\)](#)
 - [7. Video: Built-in Functions and Custom Data Types \(it_pycipydj_04_enus_07\)](#)
 - [8. Video: Custom Iterators Using Special Methods \(it_pycipydj_04_enus_08\)](#)
 - [9. Video: Defining Properties on Classes \(it_pycipydj_04_enus_09\)](#)
 - [10. Video: Defining Properties Using Decorators \(it_pycipydj_04_enus_10\)](#)
 - [11. Video: Class Methods \(it_pycipydj_04_enus_11\)](#)
 - [12. Video: Static Methods \(it_pycipydj_04_enus_12\)](#)
 - [13. Video: Abstract Base Classes \(it_pycipydj_04_enus_13\)](#)
 - [14. Video: Exercise: Advanced Functionality in Classes \(it_pycipydj_04_enus_14\)](#)
- [Course File-based Resources](#)

1. Video: Course Overview (it_pycipydj_04_enus_01)



- *discover the key concepts covered in this course*

[Video description begins] *Topic title: Course Overview.* [Video description ends]

Hello there. Welcome to this course, Advanced Functionality Using Python Classes. My name is Janani Ravi and I will be your instructor for this course.

[Video description begins] *Your host for this session is Janani Ravi. She is a software engineer and a big data expert.* [Video description ends]

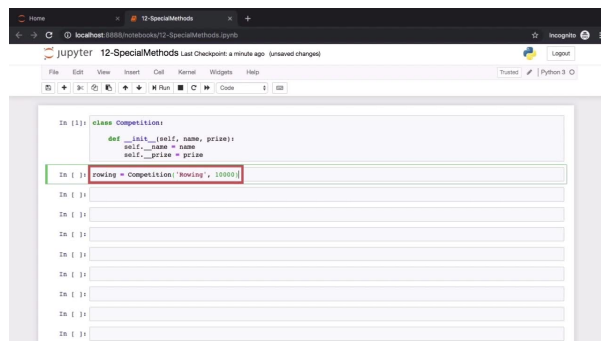
A little about myself first. I am a cofounder at Loonycorn, a studio for high-quality video content. Before founding Loonycorn, I worked at various companies, including Google, for seven years in New York and Singapore, and Microsoft, for three years in Redmond. And before that I attended grad school at Stanford.

Object-oriented programming is by now a decades-old approach to software development. But it remains as relevant and important today as it was in the 1990s. The rise of Python has not rendered object-oriented programming obsolete or irrelevant. Rather the beauty of Python's approach is that it allows an intelligent and elegant combination of the two dominant approaches to development today, object-oriented programming and functional programming.

In this course, you will learn the use of advanced aspects of object-oriented programming in Python, such as special methods which help mimic common language supported operators. An important syntactic feature that made C++ very popular in decades past was support for operator overloading. And this is precisely what you will learn how to implement in Python. You will also learn the use of iterators, class properties, and abstract base classes.

By the end of this course, you will be able to identify situations where special methods are likely to come in handy; for instance, in the creation of user-friendly string representations of objects. You will also be able to create properties for objects using decorators and implement abstract base classes in Python.

2. Video: The repr and str Special Methods (it_pycipydj_04_enus_02)



```
In [1]: class Competition:
        def __init__(self, name, prize):
            self._name = name
            self._prize = prize

In [2]: rowing = Competition('Rowing', 10000)
```

In this video, you will learn how to represent objects using customized strings.

- *represent objects using customized strings*

[Video description begins] *Topic title: The repr and str Special Methods. Your host for this session is Janani Ravi.* [Video description ends]

Now that you understand what classes are in Python and how inheritance works, we can go on to starting some special features or attributes that you can add to Python classes so that they exhibit additional useful characteristics.

[Video description begins] *A Jupyter Notebook is open. It includes a workspace comprising several code cells.* [Video description ends] These are special methods in Python, and we've actually used one of these extensively in the previous demos of this course. This is the init method which is a special method which is executed under the hood when you create or instantiate a new object of a class.

[Video description begins] *The presenter pastes some code comprising four lines in the first code cell. The following is the code per line: Line 1: class Competition;; line 2: def __init__(self, name, prize); line 3: self.__name = name, and line 4: self.__prize = prize.* [Video description ends]

Here we have the Competition class and within that we've defined the special method init. How do we know it's special? Well, because of the double underscore prefix as well as the suffix. All of the special methods in Python classes have the same prefix and suffix. This init method takes in the name and the prize as input arguments and assigns it to the instance variables, name and prize.

The init method is a special method which is invoked when you create an object of type Competition; rowing is equal to Competition. The name is Rowing, the prize money is 10000 dollars. [Video description begins] *She pastes the following code in the subsequent code cell: rowing = Competition('Rowing', 10000).* [Video description ends]

Under the hood, the init method has been invoked and the coordinate has been executed. This has resulted in the creation of the Competition instance or object. When you print out the rowing variable, you can see the memory location in Python at which this object is located.

[Video description begins] *She pastes the following code in the subsequent code cell: print(rowing). And when she runs it, the following output displays below the code cell: <__main__.Competition object at 0x1030970b8>.* [Video description ends]

Now this specific bit of information, while useful in understanding that this object has a memory location where it's stored, is not that useful. What you really want is when you print out an instance of the Competition class, maybe you want to print out the name as well as the prize money associated with that instance.

Printing out the name and the prize money might be a much better way to represent an object of the class Competition rather than just the memory location. If you simply type out the object rowing to screen, you get the same information – that custom data type, Competition, that it belongs to and the memory location. You can define a special method within Python classes in order to have the classes represented in a different manner when you use them with a print statement.

[Video description begins] *She pastes some code comprising six lines in the subsequent code cell. The following is the code per line: Line 1: class Competition;; line 2: def __init__(self, name, prize); line 3: self.__name = name, line 4: self.__prize = prize, line 5: def __repr__(self); and line 6: return "({}, {})".format(self.__name, self.__prize).* [Video description ends]

Here we've defined the Competition class. In addition to the init special method that we're familiar with, I've defined a new special method here that Python understands. This is the repr special method. We know it's a special method because of the double underscore prefix and suffix. The repr method determines how an object is represented when you print this object out to screen using the print function.

The repr method should return a string representation of your object. Now what this string representation should be, that's up to you. Here I've basically said that a Competition object should print out its name and prize within parentheses. Let's instantiate an object of the class Competition. [Video description begins] *She pastes the following code in the subsequent code cell: archery = Competition('Archery', 8000).* [Video description ends]

This is an archery competition with a prize money of 8000 dollars, and I assign this to the

archery variable. Now when I pass in this archery object to a print function and print out what this object is all about, you'll see that I get the string representation that I've specified within the repr special function. [Video description begins] *She pastes the following code in the subsequent code cell: print(archery). And when she runs the code, the following output displays below the code cell: ('Archery', 8000).* [Video description ends]

Within parentheses, we've specified the name of the competition, Archery, and also the prize money of 8000 dollars. It's also possible to see what representation is used for an object by using a built-in function which is named repr. [Video description begins] *She pastes the following code in the subsequent code cell: repr(archery).* [Video description ends]

Pass in archery to this built-in repr function, and it'll give you the string representation that we have defined for this object. [Video description begins] *The following output displays below the code cell: "('Archery', 8000)".* [Video description ends]

In Python, you can convert any data type to its string representation by passing in that data type to an str built-in function; str(archery) will use the string representation that you have defined in the repr special function. [Video description begins] *She pastes the following code in the subsequent code cell: str(archery).* [Video description ends]

Let's go into some details of how Python works. When you pass in an object to a print function in Python, under the hood, print actually invokes the str function. So it invokes the str function, passes in that object, and prints out the string representation of that object. When you pass in an object to the str built-in function, this looks for a string representation of the object first in a special method called str, double underscore str followed by a double underscore.

Now we haven't encountered or even worked with the special method, and we haven't defined this for the Competition class. When it can't find the str special method, it looks for the repr special method. And we have the repr special method defined, that is invoked and that's what prints out the string representation of this archery object.

[Video description begins] *A banner comprising two lines of text displays at the bottom of the screen. The following is the text per line: Line 1: str(archery), line 2: Looks for a string representation in the __str__() then the __repr__() function.* [Video description ends]

Let's redefine the Competition class once again with some additional special methods. We know the init.

[Video description begins] *She pastes some code comprising twelve lines in another code cell. The following is the code per line: Line 1: class Competition:, line 2: def __init__(self, name, country, prize):, line 3: self.__name = name, line 4: self.__country = country, line 5: self.__prize = prize, line 6: def get_name_country(self):, line 7: return '{} {}'.format(self.__name, self.__country), line 8: def __repr__(self):, line 9: return "Competition: {} held in {}, prize: {}".format(self.__name, self.__country, self.__prize), line 10: .format(self.__name, self.__country, self.__prize), line 11: def __str__(self):, and line 12: return '{} - {}'.format(self.get_name_country(), self.__prize).* [Video description ends]

We have a getter here which says get_name_country, which returns the name as well as the country in the string format. We have the repr special method that we studied earlier. And finally we have the new special method, the string special method. The str special method in a class is a string representation of the object once again. But this is specifically used by the print and the str built-in functions.

Observe that in our current definition of the Competition class, we have the repr as well as the str special method. But the string representation of objects here are a little different in each case. The repr special method returns a string that is a sentence – Competition, competition name, held in this country, with this prize money. Whereas the str special method simply returns the name, country, dash, and the prize money.

The differences in these implementations will allow us to know when the repr method is

invoked and when the str method is invoked under the hood. I'll now instantiate an object of type Competition. This is an archery competition held in the United Kingdom, with a prize money of 7500 dollars. [Video description begins] *She pastes the following code in the subsequent code cell: archery = competition('Archery', 'United Kingdom', 7500).* [Video description ends]

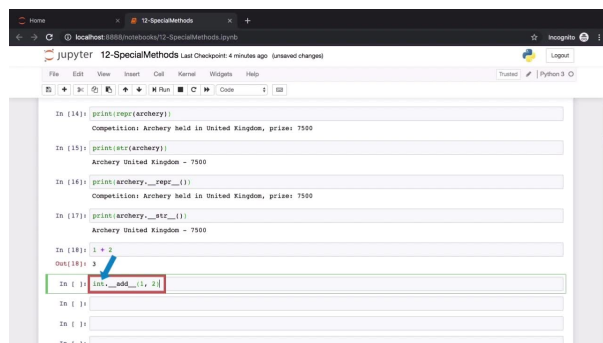
Now if I simply type out archery to screen without a print function, you'll see that we end up invoking the repr special method under the hood. The string that is printed out to screen, Competition: Archery held in United Kingdom, prize: 7500, is the string that we have defined in the repr function of this Competition class.

If you pass in the archery object to the print function, under the hood, the print function will invoke the str function on the archery object, which means it'll print out the string format that we have specified in the str special method. When you print archery, you get Archery United Kingdom - 7500. This is the string that we have specified in the str special method.

If you want the string format in the repr method, you can pass in repr(archery) into the print function. This will give you the string format in the repr method. Or you can say print (str(archery)), and this will give you the string format representation of this object in the str special method.

You can also invoke these special methods like ordinary methods on an object; archery.__repr__, we can invoke it directly and you'll get the string representation of this object. [Video description begins] *She pastes the following code in another code cell: print(archery.__repr__()).* [Video description ends] In exactly the same way, you can call archery.__str__. Notice how we're using the special methods exactly like we would ordinary methods defined within a class.

3. Video: The add Special Method (it_pycipydj_04_enus_03)



Find out how to add custom objects.

- *perform addition operations on custom objects*

[Video description begins] *Topic title: The add Special Method. Your host for this session is Janani Ravi.* [Video description ends]

Special functions, or special methods, in Python are so called because they're based on special properties on your classes. [Video description begins] *A Jupyter Notebook is open. It includes a workspace comprising several code cells. The following operation displays in one of the code cells: 1 + 2.* [Video description ends]

Whether you're working with primitive data types or complex data types or custom data types that you've created using classes, all of these are implemented as classes under the hood. For example, the integers that you're about to add right now, each of these integers belong to the int built-in class. And the fact that you're able to add them together using the + operator is

essentially because of a special method.

I'll show you how. You know that $1 + 2$ is equal to 3, as you see here on screen. The ability to be able to add two integers comes from this special method that has been declared in the int class under the hood. And the int class represents integers, the primitive data types that we are familiar with. [Video description begins] *The presenter pastes the following code in the subsequent code cell: `int.__add__(1, 2)`.* [Video description ends]

This special method on the int class is the add method. And you know it's special because it's a prefix and suffix with a double underscore. When we invoke this add special method on the int class and pass in the integers that we want to add, we'll get the same result, 3, which is equal to $1 + 2$. This is the same result that we got when we used the + operator. So essentially, when you use the + operator on any kind of data, under the hood, it invokes this add special method.

[Video description begins] *She points to the plus sign in the first code cell and to the word, add, in the subsequent code cell.* [Video description ends]

Let's see another demonstration here so that things become clearer. [Video description begins] *She pastes the following code in the subsequent code cell: `'a' + 'b'`.* [Video description ends] Here I'm invoking the + operation on two strings, a and b. In Python, you can add together strings and you get the concatenated string as a result. You can see 'a' + 'b', where 'a' and 'b' are both strings, gives you 'ab'. [Video description begins] *She points to the following output that displays below the code cell: 'ab'.* [Video description ends]

In Python, the string primitive data type, under the hood, belongs to the str class; this is a built-in class. [Video description begins] *She pastes the following code in the subsequent code cell: `str.__add__('a', 'b')`.* [Video description ends] And when you invoke the add special method, which is available on this str class, and pass in two strings, that is, 'a' and 'b', as we did before, the result will be the same – 'a' concatenated with the string 'b', which is just the string 'ab'.

Now the cool thing about special methods is the fact that you can declare the special method in other classes as well and have them respond to operators. [Video description begins] *She pastes a code comprising three lines in the subsequent code cell. The following is the code per line: Line 1: `class Savings;`, line 2: `def __init__(self, amount);`, and line 3: `self.__amount = amount.`* [Video description ends]

Let's take an example of a class called Savings, which has just one instance variable, that is, the amount of savings that you have. Let's say the savings are in dollars. [Video description begins] *She points to the word amount that forms a part of the following code fragment in the third line of code: `self.__amount`.* [Video description ends]

And we've initialized this amount using the init special method. I'll now create two objects of this Savings class, s1 and s2. Let's assume that these two Savings objects refer to the savings of two different individuals. One has saved 10000 dollars, another has saved 2000 dollars. [Video description begins] *She pastes a code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: `s1 = Savings(10000)` and line 2: `s2 = Savings(2000)`.* [Video description ends]

Now I want to calculate the total amount of savings across these two individuals. So I say, simply sum up the savings for individual one and individual two, `s1 + s2`. Do you think this will work? Well, not really. When you execute this code, what you get is a `TypeError`.

Let's take a look at the `TypeError` and you'll see that it actually makes things very explicit; unsupported operand type for +: 'Savings' and 'Savings'. It basically means that the + operator is not supported for the custom class Savings. So when you're using this Savings class, you can't use the + operator when you have Savings objects on either side of the +. Python has absolutely no idea how to add two Savings objects together.

Well, if you want to be able to add two Savings objects together, we can write some code for

that. [Video description begins] *She pastes a code comprising five lines in another code cell. The following is the code per line: Line 1: class Savings;, line 2: def __init__(self, amount);, line 3: self.__amount = amount, line 4: def __add__(self, other);, and line 5: return self.__amount + other.__amount.* [Video description ends]

We've already discussed the fact that in order to support the + operator for a certain data type, you need to implement the add special method. And that's exactly what we've done here in this redefined Savings class. The Savings class here has two special methods. One is the init, which initializes the amount in Savings, and the other is the add.

Observe the input arguments that I have passed into this add special method. The first is, of course, the self input argument, and another is a single input argument called other. The way this add special method works is that it takes as an input argument another instance of the Savings class. [Video description begins] *She points to the word other in the fourth line of code.* [Video description ends]

And then it adds the amount in this current instance of the Saving class to the amount in the other instance of the Saving class and returns the sum. [Video description begins] *She points to the word amount that appears first in the fifth line of code and then to the word amount that appears subsequently in the same line of code.* [Video description ends] Observe what this function returns, self.__amount + other.__amount. This is an add operation that allows us to add the amounts of two Savings objects.

Now that we have defined the Savings class to include this add special method, let's see this in action. I'm going to instantiate two objects of this new Savings class; so a savings of 10000 for individual one and 2000 dollars for individual two, stored in the variables s1 and s2.

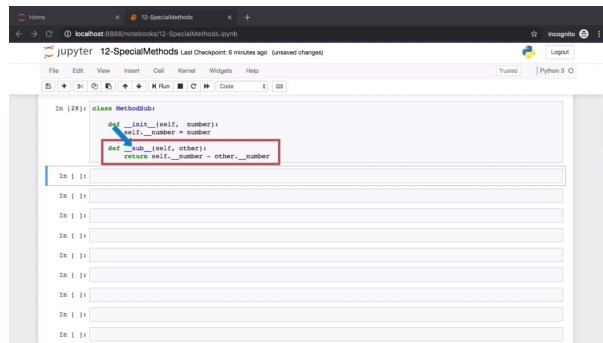
Now if you want to add up the savings of these two individuals together, we can simply invoke s1 + s2. And you will find that this does not throw an error. This works, and it'll sum up the amounts of the individual savings and give you the result 12000, which is the sum of 10000 + 2000. Let's discuss what exactly happens under the hood here when we do this.

[Video description begins] *A banner displaying the following two lines appears at the bottom: Line 1: s1.__add__(s2) and line 2: The __add__() special function is invoked on the s1 object.* [Video description ends]

When we have two objects, the add special function is invoked on the first object that you pass in. That is, we invoke s1.add. The special function is invoked on the object to the left side of the addition operator. The second object here, s2, is passed in as an input argument to this add special function. [Video description begins] *She highlights s2 in the operation s1 + s2 and also highlights the following code fragment in the fourth line of code of the Savings class definition: other);.* [Video description ends]

If you look at the add special function definition in the Savings class, you'll see that it takes in one input argument and that is s2 in this particular addition operation.

4. Video: The sub Special Method (it_pycipydj_04_enus_04)



In this video, learn how to perform subtraction operations on custom objects.

- *perform subtraction operations on custom objects*

[Video description begins] *Topic title: The sub Special Method. Your host for this session is Janani Ravi.* [Video description ends]

It's not just the plus operand that is supported using Python special methods. Other operands are also supported, such as subtraction, multiplication, division, modulo, you name it. Let's see a few examples in turn. I'm going to set up a new class here called MethodSub.

[Video description begins] *The Jupyter Notebook displays. It includes a workspace comprising several code cells. Some code comprising five lines displays in one of the code cells. The following is the code per line: Line 1: class MethodSub; line 2: def __init__(self, number); line 3: self._number = number; line 4: def __sub__(self, other); and line 5: return self._number - other._number.* [Video description ends]

This is just an example class to demonstrate how the subtraction operation can be performed on your classes. The special method that you need to implement is the sub special method. When you implement the special method within your class, objects of the MethodSub class can be subtracted. Observe that the sub special method takes in the self input argument as well as the other input argument.

This MethodSub class has just one instance variable; that is number. And within the sub special method, we perform the following operation: `self._number - other._number`. This method assumes that the other object that is passed in is also of type MethodSub. With this definition, all instances of the MethodSub class should support the subtraction operand.

And let's try this out and see. I'm going to instantiate two objects now of the MethodSub class; `num_1` and `num_2` are the variables for these objects. [Video description begins] *She pastes some code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: num_1 = MethodSub(10) and line 2: num_2 = MethodSub(8).* [Video description ends]

The value of number in `num_1` is 10 and for `num_2`, it's 8. With these two objects instantiated, let's see if the subtraction operation works. I'm going to subtract `num_2` from `num_1`. You can try subtracting `num_1` from `num_2` as well, and that should work too. [Video description begins] *She pastes the following code in the subsequent code cell: num_1 - num_2.* [Video description ends]

The result of `num_1 - num_2` gives us 2; that is 10 - 8. Now under the hood, the sub special method is invoked on the `num_1` object and `num_2` is passed in as an input argument. The way we set up the sub special method is exactly like our add special method. It works with just two objects, one object, that is, `num_1` here in this example on which the special method is invoked and the second object, `num_2` here in this example, which is passed in as an input argument.

There is something interesting that you need to know here. When you use the - operator, under the hood, Python blindly invokes the sub special function. And whatever operation you

[Video description begins] *Topic title: The mul Special method. Your host for this session is Janani Ravi.* [Video description ends]

It's pretty clear, though, that for the built-in types such as floating points, two floats can be multiplied together. And Python knows how to execute this operation; 1.0 multiplied by 2.1 gives you 2.1, this is what we'd expect.

[Video description begins] *The Jupyter Notebook is open. It includes a workspace comprising several code cells. The following code displays in one of the code cells: 1.0 * 2.1. When the presenter runs the code, the following output displays below the code cell: 2.1.* [Video description ends]

This is because you need to tell Python that a particular data type supports multiplication. For example, floating point data types in Python are types of the built-in class float, and this supports the mul special method. [Video description begins] *She pastes the following code in the subsequent code cell: float.__mul__(1.0, 2.1). She points to the word float and then to the word mul in the same code.* [Video description ends]

The mul special method supports the multiplication of many floating point values together. So you can simply invoke this mul special method on the float class, pass in a number of arguments, and it'll give you the multiplication results, multiplying all of those floats together. Let's see something new here. The mul special method, which is part of the floating point class, only supports a multiplication of floating point values. [Video description begins] *She pastes the following code in the subsequent code cell: float.__mul__(1, 2.1).* [Video description ends]

I've passed in two input arguments here to this mul special method. One of these is an integer. So we have 1 multiplied by 2.1; 1 is an integer, 2.1 is a floating point value. When you execute this code, you'll see an error, which is a TypeError. The mul special method only works with floats. When you try to invoke the mul class directly with an int rather than a float, that does not work. And the error makes this very clear – mul requires a float object but it received an int. We can do something similar with the custom classes that we define as well.

[Video description begins] *She pastes a code comprising ten lines in the subsequent code cell. The following is the code per line: Line 1: class Savings;, line 2: def __init__(self, amount);, line 3: self._amount = amount, line 4: def __add__(self, other);, line 5: return self._amount + other._amount, line 6: def __mul__(self, other);, line 7: if type(other) == int or type(other) == float;, line 8: return self._amount * other, line 9: else;, and line 10: raise ValueError('Can only multiply by int or float').* [Video description ends]

Let's go back to our Savings class, the one that we've seen previously. We have the init special function, we have the add special function, which allows us to add two Savings objects together. [Video description begins] *After highlighting the first three lines of code, she highlights the fourth and the fifth lines of code.* [Video description ends]

In addition, I've defined the mul special method, which works a little differently. It takes in two input arguments, that is self, which is there are across all class methods, as well as other. [Video description begins] *She highlights the sixth line of code.* [Video description ends] Observe the code that I have within the mul special method.

I have specified that the type of the other object should be either an integer or a floating point. [Video description begins] *Besides highlighting the sixth and the seventh lines of code, she highlights the word int and the following code fragment in the seventh line of code: float:.* [Video description ends]

So the other variable cannot hold objects of other types. The other variable should only be an integer or a floating point number. So I'm performing an explicit type check that other is of type int or of type float. If the other variable is of type integer or floating point, by the value in the other variable and return the result. [Video description begins] *She highlights the eighth line of code.* [Video description ends]

But in case the other variable holds an object of some other data type, I'll raise a ValueError and say, Can only multiply by int or float. [Video description begins] *She highlights the last two*

lines of code. [Video description ends]

Observe the implication of this. The multiplication operation on Savings objects can only work if you specify an integer or a floating point to multiply with. It won't work if you multiply two Savings objects together. And we'll see that in just a bit. [Video description begins] *Besides highlighting the last five lines of code, she highlights the word int and the following code fragment in the seventh line of code: float:.* [Video description ends]

I'll go ahead and instantiate two objects of this Savings class, Savings of 1000 and 200 stored in variables s1 and s2. The addition operation s1 + s2 should work. [Video description begins] *She pastes a code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: s1 = Savings(1000) and line 2: s2 = Savings(200).* [Video description ends]

Because, remember, this is the equivalent of invoking the add function on s1 and passing in s2 as an input argument. [Video description begins] *A banner displaying the following two lines of text displays at the bottom: Line 1: s1.__add__(s2) and line 2: The __add__() special function is invoked on the s1 object.* [Video description ends]

The add function that we've defined assumes that s2 is of type Savings, accesses the amount variable, and returns the result, which is the addition of s1 plus s2; 1200 dollars is the total savings. Now let's try and multiply two Savings objects together, s1 multiplied by s2. Now under the hood, Python will invoke the mul special method on s1. [Video description begins] *She points to the word mul in the sixth line of code.* [Video description ends]

But if you take a look at the code in the mul special method, the one that we just discussed, it can only work if what you pass in as an input argument is of type int or float. [Video description begins] *She highlights the word int and the following code fragment in the seventh line of code: float:.* [Video description ends]

And if you try to execute this code, you'll end up with a ValueError, which says, Can only multiply by int or float. Let's track step by step how we ended up with this ValueError; s1 multiplied by s2, under the hood, invoke the mul special method that we had set up in the Savings class. The way Python interprets s1 multiplied by s2 is that it invokes mul on the s1 object and passes s2 in as an input argument.

[Video description begins] *A banner comprising the following two lines of text displays at the bottom: Line 1: s1.__mul__(s2) and line 2: Here s2 passed in as an input argument. She highlights s2 in the operation, s1 * s2, and also highlights the following code fragment in the sixth line of the Savings class definition: other).* [Video description ends]

The data type for the s2 variable is of type Savings; it's a Savings object. Now if you look at the mul method within our Savings class, it expects that the other variable is of type int or float. It does not accept the other variable to be of any other data type, including of type Savings. The if check within the mul special method sees that other is not of type int or float.

Then we go into the else statement, and that's how we get this ValueError; can only multiply by int or float. [Video description begins] *After highlighting the seventh line of code in the Savings class definition, she highlights the last two lines of code in the same definition. Next, she highlights the ValueError statement in the output that displays.* [Video description ends]

What will work, though, is if you try and multiply our savings by the integer 3; s1 multiplied by 3. [Video description begins] *A banner containing the following two lines of text displays at the bottom: Line 1: s1.__mul__(3) and line 2: Here the integer 3 is passed in as an input argument.* [Video description ends]

Now what will be passed in as an input argument is the integer 3. The mul special method in the Savings class, that is invoked to perform this multiplication operation, will check that the data type for other is type int. That is completely allowed. It'll multiply the savings amount by 3 and we'll get the result. [Video description begins] *She highlights the sixth, seventh, and eighth lines*

of code in the Savings class definition. [Video description ends]

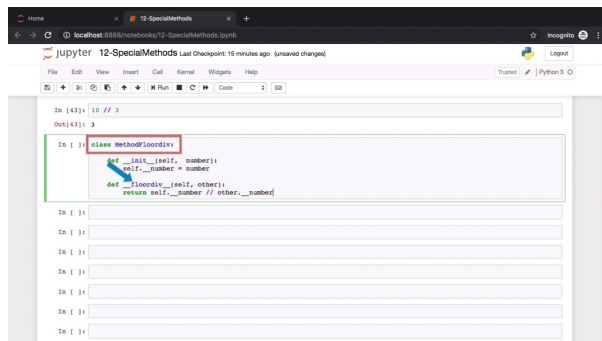
The result of multiplying s1 by 3 will multiply the savings amount in s1, that is 1000, by the integer 3 and give us 3000. And that's what you see here on screen. You can try and multiply s1 by a floating point number, s1 multiplied by 3.1. That will work; s1 multiplied by 3.1 will pass in the floating point 3.1 as an input argument to the mul special method, and that is completely valid.

[Video description begins] *A banner comprising the following two lines of text displays at the bottom: Line 1: s1.__mul__(3.1) and line 2: Here the float 3.1 is passed in as an input argument.* [Video description ends]

For practice, maybe you should try out some other combinations here; 3 multiplied by s1. [Video description begins] *A banner comprising the following two lines of text displays at the bottom: Line 1: 3 * s1 and line 2: 3.__mul__(s1).* [Video description ends] Now this will invoke the mul function on 3, which is of type integer, and pass in s1 as an input argument to the mul function on the integer class.

Now the integer class is the int built-in class in Python. It comes along with your Python installation. And it has no idea about the Savings custom class that you have created here. Because it has no idea what the Savings object is all about, it has no way to perform the multiplication, 3 multiplied by s1.

6. Video: Special Methods for Other Operations (it_pycipydj_04_enus_06)



Learn how to perform floor division, modulo, and power-of operations.

- *perform floor division, modulo, and power-of operations*

[Video description begins] *Topic title: Special Methods for Other Operations. Your host for this session is Janani Ravi.* [Video description ends]

You'll find that almost all of the operators that Python supports will have an equivalent special method, so that that operator can be implemented by our custom classes. For example, the operator that we are implementing here is the floor division operator; 10 with two forward slashes and a 3 performs floor division. That is it'll perform the division and return only the integral portion of the result; 10 divided by 3 is actually 3.3333.

[Video description begins] *The Jupyter Notebook is open. It includes a workspace comprising several code cells. The following operation displays in one of the code cells: 10 // 3.* [Video description ends]

The result we get is just 3. This is the floor division. Now the floor division is not a very common operator, but it might be that you want your custom classes to support floor division. Here is a class called MethodFloorDiv. [Video description begins] *The presenter highlights the two*

forward slashes in the operation [Video description ends]

And I've implemented the special floordiv method within this class in order to support floor division. [Video description begins] *She pastes a code comprising five lines in the subsequent code cell. The following is the code per line: Line 1: class MethodFloordiv; line 2: def __init__(self, number); line 3: self.__number = number; line 4: def __floordiv__(self, other); and line 5: return self.__number // other.__number. She points to the following code fragment in the fourth line of code in the MethodFloordiv class definition: floordiv.* [Video description ends]

The MethodFloordiv class here is very simple. It has just a single instance variable that is the number. [Video description begins] *She highlights the second and the third lines of code and points to the word number that appears first in the third line of code.* [Video description ends]

And we have the floordiv special method, which takes in just self and other. [Video description begins] *After highlighting the last two lines of code, she points to the word self and the word other that appear in the fourth line of code.* [Video description ends]

It expects that the other variable is of an object which has a number instance variable. And we simply perform the floor division operation, self.__number floordiv other.__number. [Video description begins] *She highlights the last line of code and points to the two forward slashes in the same line.* [Video description ends]

I'll now instantiate two objects of the class MethodFloordiv. These are the objects which are stored in the variables num_1 and num_2, and the numbers associated with these objects are 10 and 3. [Video description begins] *She pastes a code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: num_1 = MethodFloordiv(10) and line 2: num_2 = MethodFloordiv(3).* [Video description ends]

Now if you perform the floor division operation, num_1 floordiv num_2, this will under the hood invoke the floordiv special method. [Video description begins] *After pasting the following code in the subsequent cell, she highlights the last two lines of the MethodFloordiv class definition: num_1 // num_2.* [Video description ends]

And as we know, the floordiv special method will be invoked on num_1. This is the operand to the left side of the floordiv operator. [Video description begins] *A banner comprising the following two lines of text displays at the bottom: Line 1: num_1.__floordiv__(num_2) and line 2: The __floordiv__() special method will be invoked on the left operand. She points to num_1 in the following code: num_1 // num_2.* [Video description ends]

The object stored in the variable num_2 will be passed in as an input argument represented by the other variable within the floordiv special method. [Video description begins] *Besides pointing to the word other in the fourth line of code in the MethodFloordiv class definition, she points to num_2 in the following code: num_1 // num_2.* [Video description ends]

Now this operation will be executed and the result that we'll get is 10 floor division 3. And it's 3 as we know from earlier. [Video description begins] *She points to the output that displays below the code cell containing the following operation: 10 // 3.* [Video description ends] Another operator that you might work with in Python is the modulo operator. [Video description begins] *After pasting the following operation, she highlights the percent symbol in the same operation: 4 % 2.* [Video description ends]

As you can see here, the modulo operator is represented by the percent sign; 4 % 2 will give you the remainder after you've divided 4 by 2. As you know, 4 is perfectly divisible by 2. And the modulo operator should give us the remainder as 0. And yes indeed it does. Let's see a modulo operator where there is indeed a remainder; 5 % 2 will divide 5 by 2, the remainder here is 1. And the result that you get of this modulo operation is also 1.

If you want your own custom classes to implement the modulo operation, you need to implement the mod special method. [Video description begins] *She pastes the following code in*

the subsequent code cell: `int.__mod__(5, 2)`. Next she points to the following code fragment in the same code: `mod`. [Video description ends]

The built-in integer class has an implementation for the mod function, which is why you can perform the mod operation on integers. [Video description begins] She points to the code fragment `int` in the following code: `int.__mod__(5, 2)`. [Video description ends]

I'm going to specifically invoke the mod special function in the `int` built-in class. This will divide 5 by 2 and give us the remainder. And as we know, the remainder here is 1. Here is an example of how you can have your own custom class support, the modulo operator. I have set up an example class here called `MethodMod`.

[Video description begins] She pastes a code comprising five lines in the subsequent code cell. The following is the code per line: Line 1: `class MethodMod`; line 2: `def __init__(self, number)`; line 3: `self._number = number`; line 4: `def __mod__(self, other)`; and line 5: `return self._number % other._number`. [Video description ends]

It uses the `init` special function in order to initialize a number instance variable. Defining code for the mod special method will have this class support the modulo operator. [Video description begins] After highlighting the last two lines of code, she points to the percent symbol in the last line of code. [Video description ends] The mod special method here takes in just one other input argument, that is `other`. [Video description begins] She highlights the word `other` in the fourth line of code. [Video description ends]

It expects that this `other` variable has a property, `_number`. [Video description begins] She points to `_number` in the following code fragment in the last line of code: `other._number`. [Video description ends] The modulo operation is performed between the number property of the current instance and the other instance. [Video description begins] She highlights the percent symbol in the last line of code. [Video description ends]

I'll instantiate two objects of this `MethodMod` class, one object initialized with the number 10, the second object initialized with the number 3. [Video description begins] She pastes a code comprising two lines in the subsequent code cell. The following is the line per code: Line 1: `num_1 = MethodMod(10)` and line 2: `num_2 = MethodMod(3)`. [Video description ends]

Let's now perform the mod operator using these objects, `num_1 % num_2`. [Video description begins] A banner comprising the following two lines of text displays at the bottom: Line 1: `num_1.__mod__(num_2)` and line 2: The `__mod__()` special function is invoked on the left operand. [Video description ends]

The mod special function will be invoked on the left operand, that is the `num_1` object. And the right operand, `num_2`, will be passed in as an input argument to the mod special function. And we can perform a little mental arithmetic to calculate this mod operation; `10 % 3` is the remainder when 10 is divided by 3, which is equal to 1.

Another common math operator that you might want to perform is the power of – 6 to the power 2 is 6 square. [Video description begins] She pastes the following operation in the subsequent code cell: `6**2`. [Video description ends] That gives us 36. That's because integers support the power special function. On the integer class, if you invoke the `pow` special function and pass in 6, 2, it'll give you 6 to the power 2 which is 36. [Video description begins] She pastes the following code in the subsequent code cell: `int.__pow__(6, 2)`. [Video description ends]

If you want your objects to be able to support the power of operation, then you will simply implement this special method, `pow`. [Video description begins] She pastes a code comprising five lines in the subsequent code cell. The following is the code per line: Line 1: `class MethodPower`; line 2: `def __init__(self, number)`; line 3: `self.number = number`; line 4: `def __pow__(self, other)`; and line 5: `return self.number**other.number`. [Video description ends]

Here is a class called `MethodPower` that I've defined here. It has just the single instance

variable, number. The pow special method takes in one input argument, that is other, and we return self.number raised to the power other.number. [Video description begins] *She highlights the second and the third lines of code. She highlights the last two lines of code and points to the word other in the fourth line of code.* [Video description ends]

Let's see this power method in action by instantiating two objects of type MethodPower. The first has the number 10, the second has the number 3. [Video description begins] *She pastes a code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: num_1 = MethodPower(10) and line 2: num_2 = MethodPower(3).* [Video description ends]

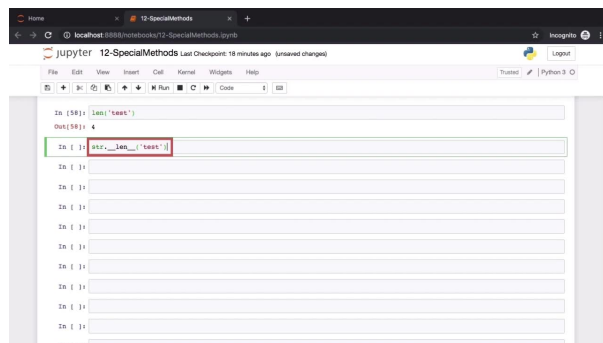
If you want to calculate the value of 10 to the power 3, you'll specify num_1** and then num_2. This will calculate 10 cube or 10 to the power 3. This is because the pow special method is invoked on the object num_1, and num_2 is passed in as an input argument.

[Video description begins] *A banner comprising the following two lines of text displays at the bottom: Line 1: num_1.__pow__(num_2) and line 2: The __pow__() special function is invoked on the left operand.* [Video description ends]

The special function is invoked on the left operand, and the right operand becomes the input argument. And as you know, 10 to the power 3 will give us 1000. Now as self-study, you can try num_2**num_1. The pow special method will be invoked on the left operand which in this case is the num_2 object. And num_1 will be passed in as an input argument, giving you 3 to the power 10. This is something that you can try out on your own as self-study.

[Video description begins] *A banner comprising the following two lines of text displays at the bottom: Line 1: num_2**num_1 and line 2: num_2.__pow__(num_1).* [Video description ends]

7. Video: Built-in Functions and Custom Data Types (it_pycipydj_04_enus_07)



Learn how to allow built-in functions to work with custom data types.

- *allow built-in functions to work with custom data types*

[Video description begins] *Topic title: Built-in Functions and Custom Data Types. Your host for this session is Janani Ravi.* [Video description ends]

It's not just mathematical operations that are supported by these special methods. There are other interesting operations as well that your custom classes can support.

[Video description begins] *The Jupyter Notebook is open. It includes a workspace comprising several code cells. The following code displays in one of the code cells: len('test').* [Video description ends] Now you might remember that you can pass in a string to the len function and it'll calculate the length for you. The length of the string is 4; test has 4 characters.

As you can see, the len is an extremely useful built-in function, and it's often used in Python. When you build your own custom data types using classes in Python, you might want these

custom data types to be supported by other built-in functions, specifically the len function. And you can make this happen. How, you might ask. Well, by the use of special method.

It so happens that the built-in string class, that is the str class, has an implementation for the len special function. Observe the prefix and the suffix double underscores. [Video description begins] *The presenter pastes the following code in the subsequent code cell: `str.__len__('test')`.* [Video description ends]

You can simply invoke `str.__len` and pass in a string, and this will calculate the length of that string. This is a special implementation for the string built-in class. [Video description begins] *She highlights the following output that displays below the code cell: 4.* [Video description ends]

It's not just the string data type that supports this len special method. We've seen that the lists that we use in Python can also be passed into the len built-in function and we get the length of a list. [Video description begins] *She pastes some code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: `some_list = [1, 4, 5, 7, 2, 3, 5]` and line 2: `len(some_list)`.* [Video description ends]

This list that we've instantiated here, `some_list`, has a length of 7. All lists in Python are custom data types, which belong to the list built-in class. The list class has an implementation for the len special method. And this is what the len built-in function uses to figure out what the length of a list is.

Now this means that any custom class which supports this len special method can be used with this len built-in function. Thus by implementing this len special method, you can have your own custom class act like native Python objects and work with built-in functions as well. Let's see how this works in practice. I'm going to have a class called Participants, which keeps a list of all of the participants, say, in a meeting, in a competition, in a school event, anything.

[Video description begins] *She pastes some code comprising seven lines in the subsequent code cell. The following is the code per line: Line 1: `class Participants;`, line 2: `def __init__(self);`, line 3: `self.__participants = []`, line 4: `def add_participant(self, name);`, line 5: `self.__participants.append(name)`, line 6: `def __len__(self);`, and line 7: `return len(self.__participants)`.* [Video description ends]

This is a list of participants for any activity. You can see within the init method of this class that it contains a list of participants. We've initialized the participants instance variable to an empty list within init. This class also defines a helper method, which allows it to add participants. And we simply add a participant by name.

The input argument to add participant is the name of the participant. And we append this name to the end of the participants list, which is our instance variable. We want the built-in len function to operate on objects of type participants and give us how many participants are there in the list. So we define the len function as a special method within the Participants class. And the return value for the len function is simply the length of the participants list within this object.

As you can see, the Participants class is fairly simple. This is for explanation purposes only. It's possible that you'll have a more complex class when you're working with Python in the real world. Observe that the len special method takes in no additional input argument; only the self variable. I'll now instantiate an object of type Participants; p is the variable name. [Video description begins] *She pastes the following code in the subsequent code cell: `p = Participants()`.* [Video description ends]

And initially, if you invoke the len function and pass in p as an input argument, we have no participants set up. This is simply the empty list; `len(p)` is 0. Let's assume that these are participants in a conference of sorts. I'm going to add a new participant to this conference. This is James. [Video description begins] *She pastes the following code in the subsequent code cell:*

`p.add_participant('James').` [Video description ends]

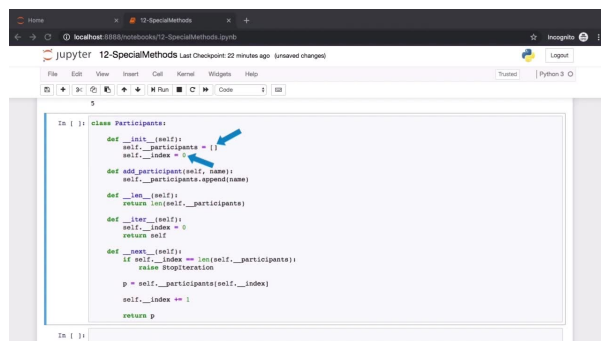
I call `p.add_participant`. And now if I invoke the `len` function on `p`, the number of participants that it returns is 1. Under the hood, the `len` function simply invokes the `len` special method on the `Participants` class in order to calculate the length of the participants. Let's go ahead and add one more participant to our conference. Emily has also participated. I call `p.add_participant`. Emily has been added.

Let's calculate the `len` of `p` now. And we get the result 2. It's pretty cool how special methods allow you to use built-in functions with your custom classes as well. Now lists in Python implement a number of different special methods which allow you to perform a different operations with lists. Calculating the length of lists is one such operation. [Video description begins] *She pastes the following code in another code cell: `list.__dict__`.* [Video description ends]

If you want to view all of the special methods and other methods that are implemented by the list built-in class, you can simply see the `dict` property of the list class. As we've seen before, this `dict` property in any class gives you a list of all of the methods within that class as well as other variable information.

Observe that the list built-in class implements all of these special methods. All of the methods that have the double underscore prefix and the suffix are special methods that lists implement. Each of these special methods add special functionality to list objects. And you can explore these on your own once you've understood the basics of how special methods work.

8. Video: Custom Iterators Using Special Methods (it_pycipydj_04_enus_08)



In this video, find out how to execute for-loops on custom data types.

- *execute for-loops on custom data types*

[Video description begins] *Topic title: Custom Iterators Using Special Methods. Your host for this session is Janani Ravi.* [Video description ends]

Before we call it a day with special methods and classes, let's explore one more very useful bit of functionality. Now we worked with lists before.

Here I've instantiated `some_list`, which is essentially just a list of integers. [Video description begins] *The Jupyter Notebook is open. It includes a workspace comprising several code cells. The following code displays in one of the code cells: `some_list = [1, 4, 5, 7, 2, 3, 5]`.* [Video description ends]

Now one of the most common operations that you perform with a list is to iterate over a list. You might iterate over a list using a for loop and then print out the contents of a list. [Video description begins] *The presenter pastes a code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: `for num in some_list:` and line 2: `print(num)`.*

[Video description ends]

Iterating over a list is very useful, and you might want your own custom classes to be such that they can be iterated over. For example, what if you want to iterate over all of the participants in your Participants class? Now iterating over a custom class is possible using a special method; not just one special method, multiple special methods put together.

[Video description begins] *She pastes a code comprising seventeen lines in the subsequent code cell. The following is the code per line: Line 1: class Participants;, line 2: def __init__(self);, line 3: self.__participants = [], line 4: self.__index = 0, line 5: def add_participant(self, name);, line 6: self.__participants.append(name), line 7: def __len__(self);, line 8: return len(self.__participants), line 9: def __iter__(self);, line 10: self.__index = 0, line 11: return self, line 12: def __next__(self);, line 13: if self.__index == len(self.__participants);, line 14: raise StopIteration, line 15: p = self.__participants[self.__index], line 16: self.__index += 1, and line 17: return p.* [Video description ends]

I've redefined the Participants class in such a way that you can now iterate over participants using a for loop. Let's take a look at all of the special methods that we have defined within Participants that make this possible. The first special method is, of course, the init method. Here is where I've initialized an empty list of participants. [Video description begins] *She highlights the second, third, and fourth lines of code. She highlights the third line of code.* [Video description ends]

Observe that I've initialized another instance variable called index to 0. [Video description begins] *She highlights the fourth line of code.* [Video description ends] The index variable here is an integer that keeps track of the current element in the Participants list. [Video description begins] *After pointing to 0 in the fourth line of code, she points to the two square brackets in the third line of code.* [Video description ends]

There are no elements in the Participants list at this point in time. So I simply initialized index to 0. You'll see how the index variable is used when we perform iteration over the participants. [Video description begins] *She highlights the third and the fourth lines of code.* [Video description ends]

The next method here within the Participants class is the add_participant, which simply appends the name of a new participant to the Participants list. This is a function that we've seen before. [Video description begins] *After highlighting the fifth and the sixth lines of code, she highlights only the sixth line of code.* [Video description ends]

The third function here is, again, a function that we are familiar with. This is a special method len, which returns the current length of participants. [Video description begins] *She highlights the seventh and the eighth lines of code.* [Video description ends]

If you want your Participants class to be an iterable, that is, you want to be able to iterate over the elements of this class, you need to define the special method iter. You can use a for loop to iterate over any object if it is an iterator. [Video description begins] *She highlights the ninth, tenth, and eleventh lines of code and also points to the following code fragment in the ninth line: iter.* [Video description ends]

How do you make an object an iterator? You implement the iter special method. When you implement the iter function, you want it to return an iterator. So what we've done here within this iter function is to initialize index to 0 so that the iteration starts at element 0 within our Participants list. [Video description begins] *On a banner appearing at the bottom, this point is written as follows: The __iter__() function returns an iterator.* [Video description ends]

And the value that this iter special method returns is self, that is, the current instance of this class. [Video description begins] *She highlights the tenth line of code.* [Video description ends] Now what makes self, that is the current instance of this class, an iterator? An object of a class can be said to be an iterator if the class implements the next special method.

So any object that responds to the next special method is an iterator object. [Video description begins] *On a banner appearing at the bottom, this point is written as follows: The class should implement __next__(). She highlights the twelfth line of code.* [Video description ends]

Now what we've done here is that the Participants class responds to next as well. We've implemented the next special method within the Participants class. Any object of the Participants class is an iterator. The next special method allows the iterator object to access the next element in the list. [Video description begins] *She highlights the last six lines of code.* [Video description ends]

So it goes through the elements sequentially starting from the very first element. What I just said here will become far clearer when you look at the code within this next special method. [Video description begins] *She highlights the last five lines of code.* [Video description ends] The next special method takes a no input argument. You simply invoke next.

The next special method will access the element available at the index variable. So if self.__index has reached the very end of the list, that is is equal to length of the Participants, then we raise the StopIteration error. [Video description begins] *After highlighting the thirteenth line of code, she points to the word index and to the word participants in the same line.* [Video description ends]

The stopiteration error is basically an indication that we've reached the end of the list, there are no further elements that you can iterate over. [Video description begins] *She highlights the fourteenth line of code.* [Video description ends] The for loop, when iterating over a list, looks for this error and then stops iteration.

In case the index value is not equal to the end of the list, in that case, we simply access the element at the current index and store it in the variable p. We then increment index by 1 and return the current element in p. [Video description begins] *After highlighting the fifteenth line of code, she points to the word index in the same line. After pointing to the sixteenth line of code, she points to the seventeenth line.* [Video description ends]

Now, in a for loop, next is called repeatedly. And each time, the variable index is incremented by 1 so that we access the next element in the list, the next element in the list, till we reach the very end of the list and StopIteration error is raised. [Video description begins] *She highlights the last five lines of code.* [Video description ends]

Take a minute and see if you understand what's happening here. [Video description begins] *After pointing to the tenth line of code, she points to the fourteenth line.* [Video description ends] We'll look at how this works with an example in just a bit. [Video description begins] *She highlights the Participants class definition.* [Video description ends]

Now that we have redefined our Participants class so that you can iterate over it using a for loop, let's instantiate Participants and add a bunch of participants in – Lily, James, Harry, Ron, and Hermione.

[Video description begins] *She pastes a code comprising six lines in the subsequent code cell. The following is the code per line: Line 1: participants = Participants(), line 2: participants.add_participant('Lily'), line 3: participants.add_participant('James'), line 4: participants.add_participant('Harry'), line 5: participants.add_participant('Ron'), and line 6: participants.add_participant('Hermione').* [Video description ends]

Now I'm going to use a for loop to iterate over my Participants class. [Video description begins] *She pastes a code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: for p in participants: and line 2: print(p).* [Video description ends]

Observe that I'm not iterating over the list that is an instance variable of participants. I'm directly iterating over the participants object of this class Participants. [Video description begins] *After highlighting the third line of code in the Participants class definition, she points to*

the word participants that appears first in the following line of code: participants = Participants(). [Video description ends]

Now I can use this for loop in this manner only because the Participants class implements the iter special method and the next special method. [Video description begins] *She points to the word iter and to the word next in the ninth and the twelfth lines of code, respectively, in the Participants class definition. [Video description ends]*

So I access one element at a time, and I'm going to print out these elements to screen. [Video description begins] *She highlights the two-line code that starts with the following line: for p in participants:. [Video description ends]* And if you execute this code, you'll have all of our participants printed out to screen. Observe that our participants custom data type now behaves exactly like a list or any other iterable.

[Video description begins] *The following names display in a sequence below the code cell: Lily, James, Harry, Ron, and Hermione. [Video description ends]*

Now this seemed to work like magic, but what exactly happened under the hood? How did iter and next together allow us to set up this for loop in this fashion? Well, this is the first step. The for loop, when you use it with a custom data type, invokes the iter special method on that data type.

[Video description begins] *On a banner that appears at the bottom, this point is written as follows: The for-loop first calls participants.__iter__(). After highlighting the code that starts with the following line, she highlights the ninth, tenth, and eleventh lines of code in the Participants class definition: for p in participants:. [Video description ends]*

This allows the for loop to get access to the iterator for that custom data type because the for loop needs an iterator to iterate over the elements. When the iter special method is invoked on the participants object, the index variable within participants is initialized to 0 and we return self, which means the iterator is the object itself. An iterator is any object that responds to the next special method.

[Video description begins] *A banner comprising the following two lines of text displays at the bottom: Line 1: An iterator is any object that responds to __next__() and line 2: The iterator here is the object itself i.e. the participants object. She highlights the tenth and the eleventh lines of code in the Participants class definition. [Video description ends]*

Now that the for loop in Python has access to an iterator, it'll simply keep invoking next on that iterator till it gets a StopIteration error. [Video description begins] *She highlights the code that starts with the following line: for p in participants:. A banner comprising the following two lines of text displays at the bottom: Line 1: The for-loop invokes __next__() and line 2: Over and over again on the iterator till it gets the StopIteration error. [Video description ends]*

Within the next special method, we first check whether the index has gone to the very end of the participants list. If it has, we raise the StopIteration error. [Video description begins] *After highlighting the thirteenth line of code in the Participants class definition, she highlights the fourteenth line of code in the same definition. [Video description ends]*

If it hasn't, we simply access the element at the current index, increment the index by 1, and return the current element. [Video description begins] *She highlights the last three lines of code in the Participants class definition. [Video description ends]*

So in the first iteration, when index is equal to 0, we get the first element in the list. [Video description begins] *She highlights the following line of code: participants.add_participant('Lily'). And besides highlighting the name Lily that displays in the output, she highlights the tenth line of code in the Participants class definition. [Video description ends]*

In the next iteration, when next is invoked once again, the index is equal to 1 and we get the

element James, and so on. [Video description begins] *She highlights the following line of code: `participants.add_participant('James')`. And besides highlighting the name James that displays in the output, she highlights the sixteenth line of code in the Participants class definition and points to the tenth line of code in the same definition.* [Video description ends]

Let's change our participants object and add one more participant. This participant is named Ginny. [Video description begins] *She pastes the following code in the subsequent code cell: `participants.add_participant('Ginny')`.* [Video description ends]

Now once again, let's use the for loop to iterate over the participants custom data type. [Video description begins] *She pastes the code comprising the following two lines in the subsequent code cell: Line 1: `for p in participants:` and line 2: `print(p)`.* [Video description ends]

And here you can see at the very end of the list, Ginny is also present and printed out. Observe that when we run the for loop once again, we start off again at the very beginning of the list. [Video description begins] *After highlighting the code that starts with the following code fragment, she points to the third line of code in the Participants class definition: `for p in participants:`* [Video description ends]

This is because of the invocation to the iter function, and the iter function initializes index to 0. [Video description begins] *She points to the tenth line of code in the Participants class definition.* [Video description ends]

If you forget this initialization, you'll find that this will throw all kinds of errors. If you try to iterate over the list more than once, you'll find that your object doesn't work the way you want it to. Resetting the index to 0 within the iter special method is extremely important. [Video description begins] *She highlights the ninth, tenth, and eleventh lines of code in the Participants class definition.* [Video description ends]

Now all of the operations that our for loop performed under the hood you can perform manually to see how the for loop works. [Video description begins] *She pastes the following code in the subsequent code cell: `iter(participants)`.* [Video description ends]

Python has available a special built-in function called iter, allowing you to access the iterator for any custom data type. We pass in an object of the participants class to the iter function, and this will give you the iterator object. And in our case here, the iterator object is the Participants class itself.

[Video description begins] *When she runs the code, the following output displays below the code cell: `<__main__.Participants at 0x103136828>`. She highlights the first line of code in the Participants class definition.* [Video description ends]

Python also have a built-in function called next, which will invoke the next special method within your object. [Video description begins] *She pastes the following code in the subsequent code cell: `next(participants)`.* [Video description ends] So calling next on participants will give you the next element. It'll start off, of course, with the very first element, Lily.

[Video description begins] *When she runs the code, the following output displays below the code cell: Lily. She highlights the word Lily in the following code fragment: `participants.add_participant('Lily')`.* [Video description ends]

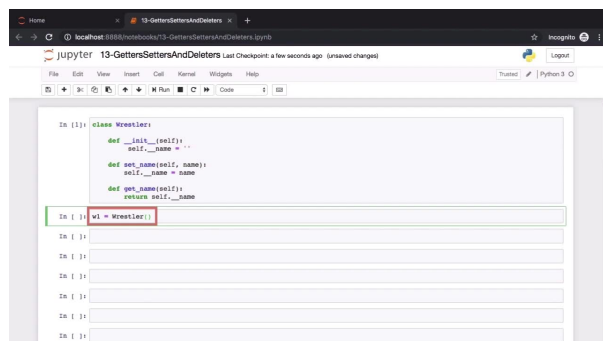
Lily is, of course, the first element in our list. Let's call next participants once again, and we'll get the next element which comes after Lily. That is the element James. Let's call next once again. And this will give you the next element, that is, Harry. And you can keep going on till you reach the very end of the list. Next, once again, will give you Ron.

We are getting close to the end of the list here. If you invoke next, you will get Hermione. And I believe there's one last participant left. Let's call next on the participants object once again, and you get Ginny. Ginny is the last element in the list. Let's try invoking next on participants once

again. Because we haven't reset the index to 0, we haven't called the iter function, you'll find that this is an error. [Video description begins] *She points to the tenth line of code in the Participants class definition.* [Video description ends]

The value of index has gone beyond the list of participants that we had, and we raised the StopIteration error. The manual set of steps that we followed here by calling iter and then next over and over again is exactly what a for loop does.

9. Video: Defining Properties on Classes (it_pycipydj_04_enus_09)



Learn how to define properties on classes for easy use.

- *define properties on classes for intuitive use*

[Video description begins] *Topic title: Defining Properties on Classes. Your host for this session is Janani Ravi.* [Video description ends]

Now typically when you set up a Python class, it can have several instance variables. And you set up getter and setter functions for your instance variables; getter functions to access the values of instance variables and setter functions to update values of instance variables.

Here is an example of a class Wrestler. [Video description begins] *The Jupyter Notebook is open. It includes a workspace comprising several code cells. A code comprising seven lines displays in the first code cell. The following is the code per line: Line 1: class Wrestler;; line 2: def __init__(self); line 3: self.__name = ''; line 4: def set_name(self, name); line 5: self.__name = name; line 6: def get_name(self); and line 7: return self.__name.* [Video description ends]

Now this is, of course, a very simple class for the purposes of our demo. In the real world, your classes will be far more complex. Within the init method of the Wrestler class, I have defined a name instance variable, which I've set to the empty string. [Video description begins] *After highlighting the second and the third lines of code, the presenter points to the two single quotation marks in the third line.* [Video description ends]

Now if you remember, the attributes of a class, that is, its properties, can be accessible from outside the class unless you prefix it with a double underscore. [Video description begins] *She points to the double underscore in the third line of code.* [Video description ends]

I don't want anyone inadvertently modifying the name attribute of the Wrestler class, which is why I have the double underscore. And I have getters and setters for this name instance variable, get_name and set_name. [Video description begins] *After highlighting the last four lines of code, she points to set_name in the fourth line and to get_name in the sixth line of code.* [Video description ends]

I'm going to instantiate an object of the Wrestler class and assign it to variable w1. [Video description begins] *She pastes the following code in the subsequent code cell: w1 = Wrestler().*

[Video description ends] With this object instantiated, I'm going to set the name of this Wrestler to Steve and invoke the set_name setter function here, passed in the name. [Video description begins] *She pastes the following code in the subsequent code cell:*
`w1.set_name('Steve').` [Video description ends]

And this will update the name instance variable. I can now call get_name and it'll return 'Steve'. [Video description begins] *She pastes the following code in the subsequent code cell and runs it: w1.get_name().* [Video description ends]

Now this name attribute of this w1 class can't be accessed directly. So if I try to access w1.name, that is an error. This is because I have specified name using the double underscore. Let's try accessing __name for this w1 object. [Video description begins] *She pastes the following code in the subsequent code cell and runs the same: w1.__name.* [Video description ends]

That is an error as well. This is because if you prefix the instance variable of a class with a double underscore, Python will garble it so that you can't inadvertently access it. If you look at the dictionary associated with this w1 object, you'll see the value for the name instance variable is stored as _Wrestler__name. That is the key which has the value Steve.

[Video description begins] *She pastes the following code in the subsequent code cell: w1.__dict__. And when she runs the same, the following output displays below the code cell: {'_Wrestler__name': 'Steve'}.* [Video description ends]

Now if you try to access the name of the wrestler in this object, w1, by specifying _Wrestler__name, that will give you Steve. So it is possible for you to access the name attribute if you know how Python has structured it. Now with that in mind, let's move on. I'm going to redefine the Wrestler class here.

[Video description begins] *She pastes a code comprising ten lines in the subsequent code cell. The following is the code per line: Line 1: class Wrestler;, line 2: def __init__(self);, line 3: self.__name = ''; line 4: def set_name(self, name);, line 5: print('setter method called');, line 6: self.__name = name, line 7: def get_name(self);, line 8: print('getter method called');, line 9: return self.__name, and line 10: name = property(get_name, set_name).* [Video description ends]

Now for every instance variable, you'll have a getter and the setter. And it's kind of tedious accessing a particular instance variable, updating it using the setter, accessing it using a getter. [Video description begins] *She points to set_name and get_name in the fourth and the seventh lines of code, respectively.* [Video description ends]

There is a way you can make things easier for yourself by using properties in Python. What exactly is a property? Well, let's see. I have the same Wrestler class with just the name instance variable, and I have a set_name setter, a get_name getter. [Video description begins] *She highlights the portion of the code starting at the fourth line and ending at the ninth line.* [Video description ends]

Within the setter and the getter, I've printed out a simple print statement, setter method called, getter method called. This is just so that I know when these methods are invoked under the hood. Observe at the very bottom here, I have defined a new property for this Wrestler class. The name of this property is name. I instantiated it as a property, and I've passed in the functions get_name and set_name.

The property function that I've invoked here allows me to create a property for this Wrestler class. What exactly is a property? We'll see how it's used in just a bit so things will become clearer. The property function is invoked using four input arguments. The four input arguments are fget, that is the getter function for the property, fset is the setter function for the property.

These are the two arguments that I've passed in to the property function here in this example. The property function also takes in additional arguments if you want to specify them; fdel is the

deleter function for the property, and doc is the documentation – any string documentation that you want to specify for the property.

Now what exactly is a Python property? It's something cool and it makes working with Python objects very straightforward. [Video description begins] *She highlights the following code fragment in the last line of code: `property(get_name, set_name)`.* [Video description ends] Now it's hard to define what it is till you see it in action. And that's what we'll do now. [Video description begins] *She pastes the following code in the subsequent code cell: `w = Wrestler()`.* [Video description ends]

I'm going to instantiate an object of the `Wrestler` class and store it in the `w` variable. When you declared a property for your class, you can access the getter and the setter of your property using intuitive syntax such as this – `w.name = 'Kart'`. Now under the hood, Python will call the setter variable for the name property. [Video description begins] *She points to `set_name` in the fourth line of code in the `Wrestler` class definition.* [Video description ends]

When you execute this bit of code, you'll see that your setter method has been invoked. We've specified that the setter for the name property is the `set_name` function, and the `set_name` function was called under the hood. [Video description begins] *When she runs the code `w.name = 'Kart'`, the following output displays below the code cell: `setter method called`.* [Video description ends]

Observe how intuitive it is to set the name of the wrestler using the equal to sign, just like you would with other variables. Now if I'd simply invoke `w.name`, under the hood, the getter method will be called. And this will return the name of the wrestler, `Kart`. We define name as a property of the `Wrestler` class, and the getter method for this property is the `get_name` function. [Video description begins] *She highlights the seventh, eighth, and ninth lines of code in the `Wrestler` class definition.* [Video description ends]

Simply accessing the name property without the equal to sign invoked the getter method. Let's redefine the `Wrestler` class once again and add to the name property.

[Video description begins] *She pastes a code comprising thirteen lines of code in the subsequent code cell. The following is the code per line: Line 1: `class Wrestler`;; line 2: `def __init__(self)`;; line 3: `self.__name = ''`; line 4: `def set_name(self, name)`;; line 5: `print('setter method called')`; line 6: `self.__name = name`; line 7: `def get_name(self)`;; line 8: `print('getter method called')`; line 9: `return self.__name`; line 10: `def del_name(self)`;; line 11: `print('deleter method called')`; line 12: `del self.__name`; and line 13: `name = property(get_name, set_name, del_name)`.* [Video description ends]

Here I have the `init` method as before, I have the `set_name` and `get_name`, our setter and getter. I have a deleter function as well. I've defined a `del_name` function within the `Wrestler` class, which simply deletes the name property. [Video description begins] *She highlights the tenth, eleventh, and twelfth lines of code.* [Video description ends]

Then when I create the name of property using the property function, I have specified `fget`, `fset`, and `fdel`. This is the same property as we saw earlier except that I now have the ability to delete the memory associated with this property. [Video description begins] *Besides highlighting the last line of code. she highlights the following code fragment in the same line: `del_name`.* [Video description ends]

Deleting this property will invoke the deleter function, which in our case is `del_name`. We'll see this property in action now. I'm going to instantiate `Wrestler` object, store it in the variable `w`. [Video description begins] *She pastes the following code in the subsequent code cell: `w = Wrestler()`.* [Video description ends]

Let's set a name for this wrestler. I'm going to call this wrestler `Kane`, and this will call the setter method under the hood. [Video description begins] *After pasting the following code in the subsequent code cell, she highlights the fourth, fifth, and sixth lines of code in the `Wrestler`*

class definition: w.name = 'Kane' [Video description ends]

By accessing the name property that we defined and using the equal to sign, we end up calling the setter method. And we know this because we have a print statement in there, setter method called. [Video description begins] *She points to the output that displays when she runs the following code: w.name = 'Kane' [Video description ends]*

Invoking the name property without the assignment operator or the equal to sign will invoke the getter method, and Kane is returned to us. [Video description begins] *After pasting the code w.name in the subsequent code cell, she highlights the same and also the seventh, eighth, and ninth lines of code in the Wrestler class definition. And when she runs the code, an output containing the following two lines of text displays below the code cell: Line 1: getter method called and line 2: 'Kane' [Video description ends]*

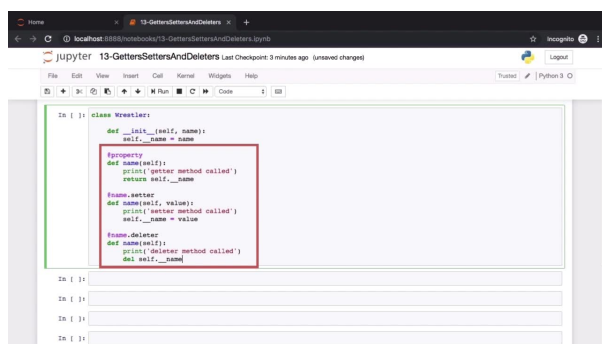
Now thanks to our deleter function, you can delete w.name as well. Using the del command on w.name will invoke the deleter function under the hood, which in our case is the del_name function. [Video description begins] *She highlights the tenth, eleventh, and twelfth lines of code in the Wrestler class definition. [Video description ends]*

Just like in the case of other deleted variables, if you try to access the name property after having deleted it, you'll get an error. Here is an AttributeError, the Wrestler object has no attribute '_Wrestler__name'. Observe how Python knows that the name property is associated with the __name instance variable. [Video description begins] *After pasting the following code in the subsequent code cell, she runs the same: w.name. [Video description ends]*

You can see the stack trace here; w.name tries to access the getter method, that is, get_name, which tries to access the instance variable __name. [Video description begins] *She highlights the seventh, eighth, and ninth lines of code in the Wrestler class definition. [Video description ends]*

That's no longer available thanks to our deleter function from earlier, and that's why we get this error. [Video description begins] *She highlights the tenth, eleventh, and twelfth lines of code in the Wrestler class definition. [Video description ends]*

10. Video: Defining Properties Using Decorators (it_pycipydj_04_enus_10)



In this video, you will learn how to define properties using a simpler syntax.

- *define properties using a simpler syntax*

[Video description begins] *Topic title: Defining Properties Using Decorators. Your host for this session is Janani Ravi. [Video description ends]*

As you've seen earlier, creating properties so that we access our instance variables in a more

natural style is very, very useful. In fact, you can make creating properties much easier when you work with decorators in Python. Decorators in Python are preceded by an at-the-rate sign. Decorators allow you to extend the functionality of functions without directly modifying the code associated with that function.

Let's see how decorators make defining your properties easier. Here we are in the Wrestler class that we worked with before, and this Wrestler has a name. That's the only instance variable of this class, that's the only property.

[Video description begins] *The Jupyter Notebook is open. It includes a workspace comprising several code cells. A code comprising fifteen lines displays in one of the code cells. The following is the code per line: Line 1: class Wrestler:, line 2: def __init__(self, name):, line 3: self.__name = name, line 4: @property, line 5: def name(self):, line 6: print('getter method called'), line 7: return self.__name, line 8: @name.setter, line 9: def name(self, value):, line 10: print('setter method called'), line 11: self.__name = value, line 12: @name.deleter, line 13: def name(self):, line 14: print('deleter method called'), and line 15: del self.__name. The presenter highlights the second and the third lines of code.* [Video description ends]

I want to define a property called name just as before. And observe that I have three methods within this class and all of these methods have the same name, that is, name. [Video description begins] *After highlighting the portion of the code that starts at line 4 and ends at line fifteen, she points to the word name in the fifth, ninth, and thirteenth lines, respectively.* [Video description ends]

Each of these three methods correspond to a getter, a setter, and a deleter for the name property. And let's see how configure this. Observe the first function here called name, which takes in only the self as the input argument. [Video description begins] *After highlighting the portion of the code starting at line 4 and ending at line 7, she points to the word name and then to the word self in the fifth line of code.* [Video description ends]

Observe the decorator for this function, the @property. When you specify the @property decorator, what you're essentially telling Python here is that this is the method that you want to invoke when the name property is accessed. This is the getter method. [Video description begins] *After pointing to the fourth line of code, she points to the word name in the fifth line.* [Video description ends]

The getter method has the decorator, which is simply @property. Let's look at the second function named name here. [Video description begins] *She highlights the portion of the code starting at line 8 and ending at line 11.* [Video description ends] It takes in self and a value. Value is an additional input argument. This is our setter function. And this is the setter function for the name property. [Video description begins] *After pointing to the word self and the word value in the ninth line of code, she points to the word name in the same line.* [Video description ends]

So the decorator that you need to specify is @name.setter. The @name indicates the name of the property with which this setter is associated. The .setter indicates to Python that this is the set function. This is the function to be invoked when we want to update the value of the name property.

And finally, we have the last function here with the same name, which only takes in the self input argument. This is our deleter function. [Video description begins] *After highlighting the last four lines of code, she points to the word self in the thirteenth line.* [Video description ends]

And observe how we decorate this function indicating that this is the deleter function, @name.deleter. This decorator indicates that this is the deleter for the name property. [Video description begins] *She highlights the last four lines of code.* [Video description ends]

When you use these decorators, Python will interpret all of these individual methods correctly. You don't have to set up a special name property as we did earlier. [Video description begins]

She points to the fourth line of code and to the following code fragment in the eighth and the twelfth lines of code, respectively: @name. [Video description ends]

Working with decorators might seem a little difficult at first. But with practice, you'll find that this is a more intuitive syntax. Let's instantiate an object of type `Wrestler` and assign it to the `w` variable. [Video description begins] *She pastes the following code in the subsequent code cell:* `w = Wrestler('Adam')`. [Video description ends]

When I access `w.name`, this will invoke the getter function for the `name` property. [Video description begins] *She highlights the portion of the `Wrestler` class definition starting at line 4 and ending at line 7.* [Video description ends] And the getter is that method that is decorated using `@property`. When you execute this code, observe that getter method called is printed out to screen. [Video description begins] *The following text displays below the text 'getter method called' in the output: 'Adam'.* [Video description ends]

This has invoked the getter function decorated using `@property`. Let's update this `name` property associated with this class by using the assignment operator `w.name = 'John'`. It was previously `Adam`. Now this assignment operator automatically, under the hood, will invoke the setter function. [Video description begins] *She points to equals in the following code: `w.name = 'John'`.* [Video description ends]

And the setter function is the one they created using `@name.setter`. [Video description begins] *She highlights the portion starting at line 8 and ending at line 11 in the `Wrestler` class definition.* [Video description ends] Under the hood, this code will invoke our setter method. And you can see that it prints out to screen `setter method called`. These print statements are useful to see what's going on behind the scenes. [Video description begins] *She points to line 10 in the `Wrestler` class definition.* [Video description ends]

And finally, let's go ahead and delete the `name` property, `del w.name`. This will invoke the code that is decorated using `@name.deleter`. And the code in our deleter function will release the memory associated with the `name` instance variable. And we get confirmation that our deleter method has indeed been called thanks to our print statement. Now if you try accessing the `name` of property after it has been deleted, this predictably results in an error because we tried to access the `__name` instance variable when the memory associated with it has been released.

[Video description begins] *When she runs the code `w.name`, the output that the application displays includes the following: `AttributeError: 'Wrestler' object has no attribute '_Wrestler__name'`.* *She also highlights `self.__name` in the the following line of code included in the output: `return self.__name`.* [Video description ends]

We'll complete our discussion of properties specified using decorators by looking at one last example.

[Video description begins] *She pastes a code comprising twenty eight lines in the subsequent code cell. The following is the code per line: Line 1: `class Wrestler:`, line 2: `def __init__(self, name, age):`, line 3: `self.__name = name`, line 4: `self.__age = age`, line 5: `@property`, line 6: `def name(self):`, line 7: `print('name getter method called')`, line 8: `return self.__name`, line 9: `@name.setter`, line 10: `def name(self, value):`, line 11: `print('name setter method called')`, line 12: `self.__name = value`, line 13: `@name.deleter`, line 14: `def name(self):`, line 15: `print('name deleter method called')`, line 16: `del self.__name`, line 17: `@property`, line 18: `def age(self):`, line 19: `print('age getter method called')`, line 20: `return self.__age`, line 21: `@age.setter`, line 22: `def age(self, value):`, line 23: `print('age setter method called')`, line 24: `self.__age = value`, line 25: `@age.deleter`, line 26: `def age(self):`, line 27: `print('age deleter method called')`, and line 28: `del self.__age`.* [Video description ends]

We'll work with the `Wrestler` class once again. But this `Wrestler` class has two instance variables associated with it. Observe the `init` function in this `Wrestler` class. And you can see that there are two instance variables, the `__name` and the `__age` instance variables. [Video description begins] *She highlights the third and the fourth lines of code.* [Video description

ends]

The remaining functions defined within this class are the getters, setters, and deleters for the name and the age properties. Observe that the functions associated with the name property all have the same name, and that name is name. [Video description begins] *She highlights the word name that displays in the sixth line, the tenth line, and the fourteenth line, respectively.* [Video description ends]

The first function here decorated using @property is the getter for the name property. We have a print statement 'name getter method called'. [Video description begins] *After highlighting the portion of code starting at line 5 and ending at line 8, she points to line 7.* [Video description ends]

The second function is the setter for the name property. It takes in value as an additional input argument. And we print within it 'name setter method called'. Observe that the decorator for this setter function is @name.setter, indicating that it's a setter for the name property. [Video description begins] *After highlighting the word value that displays in line 10, she points to line 11.* [Video description ends]

The third function here is the deleter for the name property decorated using @name.deleter. The remaining three methods here within this Wrestler class are associated with the age property. This is a Wrestler class with two properties. Observe that all of the functions have the same name, and that name is age. [Video description begins] *She highlights the last twelve lines of code. She highlights the word age that displays in lines 18, 22, and 26, respectively.* [Video description ends]

The different decorators for each of these methods tell us whether it's a getter, setter, or a deleter. Observe the first age function here, within which we print out 'age getter method called', is our getter, and it's decorated using @property. [Video description begins] *She highlights the portion of the code starting at line 17 and ending at line 20.* [Video description ends]

The function after that one is also called age. It takes in an additional input argument, value, and that is the setter method for the age property. [Video description begins] *She points to the word value that displays in line 22.* [Video description ends]

The decorator for the setter for the age property is @age.setter, indicating age is the property and this is the setter method. And finally, the last method here is the deleter for the age property decorated using @age.deleter. Once this class has been defined, let's instantiate a wrestler whose name is Mark and whose age is 25.

If you access the name property, it'll invoke the name getter function, and it'll return 'Mark'. [Video description begins] *She pastes the following code in the subsequent code cell: w = Wrestler('Mark', 25).* [Video description ends]

If you want to update the name of this wrestler, you'll simply set up an assignment statement. I've updated the name of the wrestler to Joe, and this will call the name setter method.

[Video description begins] *She pastes the following code in the subsequent code cell: w.name. Next, she highlights the portion of the Wrestler class definition starting at line 5 and ending at line 8. And after running w.name, she pastes the following code in the subsequent code cell: w.name = 'Joe'. Next, she highlights the portion of the Wrestler class definition starting at line 9 and ending at line 12. And when she runs w.name = 'Joe', the following output displays below the code cell: name setter method called.* [Video description ends]

If you want to delete this property altogether, you can invoke the del command on the name property, and it'll call the name deleter. [Video description begins] *She pastes the following code in the subsequent code cell: del w.name. She also highlights the portion of the portion of the Wrestler class definition starting at line 13 and ending at line 16. And when she runs the*

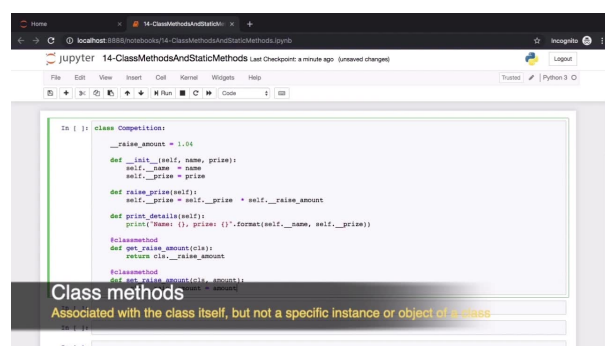
code, the following output displays below the code cell: name deleter method called. [Video description ends]

This Wrestler object has one more property that is the age property. [Video description begins] She pastes the following code in the subsequent code cell: `w.age`. [Video description ends] And you can see that the age of our Wrestler is 25. The age getter method is invoked to give us this information.

You can call the setter for the age property by using the assignment operator. [Video description begins] She pastes the following code in the subsequent code cell: `w.age = 26`. And when she runs the code, the following output displays: age setter method called. [Video description ends]

We've updated the age of our wrestler to 26 here. And finally, you can delete the age property as well using the `del` command. [Video description begins] She pastes the following code in the subsequent code cell: `del w.age`. [Video description ends]

11. Video: Class Methods (it_pycipydj_04_enus_11)



During this video, you will learn how to work with class methods to access and update class state.

- work with class methods to access and update class state

[Video description begins] Topic title: Class Methods. Your host for this session is Janani Ravi. [Video description ends]

When we discussed variables in classes earlier, we discussed the difference between instance variables and class variables. Now so far all of the methods that we've been working with are methods that are associated with objects of a class, methods that you would invoke on class objects.

You can think of all of the methods that we've been working with so far, whether they are special methods or regular getters, setters, any other method as instance methods. Instance methods have access to instance variables in a class and they can update the state of these instance variables as well. [Video description begins] According to a banner that displays at the bottom, instance methods can reference and update instance state. [Video description ends]

When you define methods within the class, there are two other kinds of methods that exist. And those are the methods that we'll discuss here, class methods and static methods. We'll go back to a class that we are familiar with. This is the Competition class, and we're going to have this Competition class include class methods.

Class methods are methods that are associated with the class itself but not with a specific object of a class. This is the main difference between class methods and all of the instance methods that we've seen so far. Class methods are associated with a class, not with an object or

instance. Let's take a look at this Competition class here and see exactly what's going on.

[Video description begins] *A Jupyter Notebook is open. It includes a workspace comprising several code cells. A code comprising fifteen lines displays in the first code cell. The following is the code per line: Line 1: class Competition:, line 2: __raise_amount = 1.04, line 3: def __init__(self, name, prize):, line 4: self.__name = name, line 5: self.__prize = prize, line 6: def raise_prize(self):, line 7: self.__prize = self.__prize * self.__raise_amount, line 8: def print_details(self):, line 9: print("Name: {}, prize: {}".format(self.__name, self.__prize)), line 10: @classmethod, line 11: def get_raise_amount(cls):, line 12: return cls.__raise_amount, line 13: @classmethod, line 14: def set_raise_amount(cls, amount):, and line 15: cls.__raise_amount = amount.* [Video description ends]

This Competition class has a class variable called `__raise_amount`, which is set to 1.04. This is the amount by which the prize money can be raised. Observe that the `__raise_amount` is a class variable and not an instance variable. Within the `init` method, we've defined two instance variables, the name of the competition and the prize money associated with the competition. [Video description begins] *The presenter highlights the fourth and the fifth lines of code.* [Video description ends]

These variables are defined on the `self` object; `self.__name` and `self.__prize`, these are instance variables. Now we have a method here called `raise_prize`. [Video description begins] *She highlights the sixth and the seventh lines of code.* [Video description ends]

Observe that the first argument that's passed in to this method is the `self` object. This means this is an instance method associated with the object `self`; `raise_prize` takes in the current value of the prize money, multiplies it by the `__raise_amount`, which is a class variable, and assigns the new value of prize money.

[Video description begins] *She highlights the following code fragment in the seventh line of code: self.__prize * self.__raise_amount. And after highlighting the second line of code, she highlights the following code fragment in the seventh line: self.__prize.* [Video description ends]

The default increase here is 4%. We have another instance method called `print_details`. [Video description begins] *She highlights the eighth and the ninth lines of code.* [Video description ends]

The first argument that it takes in is `self`, that is the object itself. That's how we know this is an instance method. The `print_details` instance method simply prints out the details of the competition. The remaining two methods are class methods. And the way Python identifies class methods as being class methods and not instance methods is by the use of the decorator `@classmethod`. The two class methods are `get_raise_amount` and `set_raise_amount`.

Let's take a look at the `get_raise_amount` class method first. [Video description begins] *She highlights the tenth, eleventh, and twelfth lines of code.* [Video description ends] Observe that the first input argument I've named, `cls`. That's because `get_raise_amount`, the class method, the first input argument, is a reference to the class. [Video description begins] *She highlights the following code fragment in the eleventh line of code: cls.* [Video description ends]

This is a reference to the class `Competition` and not to an object or an instance of the class `Competition`. Class methods can access the state within a class, not within an object or an instance, which means it can access the class variable `__raise_amount`. So we use the `cls` reference to access the `__raise_amount` and we return that from the `get_raise_amount` class method. [Video description begins] *She highlights the twelfth line of code.* [Video description ends]

The second class method here, annotated using `@classmethod`, is the `set_raise_amount`. [Video description begins] *She points to the following code fragment in the twelfth line of code: cls.* [Video description ends] In addition to the first input argument, which is a reference to the class, the second input argument is the amount. This is the new amount that we want to specify

for the `__raise_amount` class variable.

So remember, class methods have access to the state of the class; `__raise_amount` is a class variable associated with the `Competition` class. [Video description begins] *She highlights the second line of code.* [Video description ends] We change the `__raise_amount` and set its new value.

With our class defined, we're ready to see how this works in practice. [Video description begins] *She pastes the following code in the subsequent code cell: `sprint = Competition('Sprint', 10000)`.* [Video description ends]

I'm going to instantiate an object of the class `Competition`. This is a Sprint competition with the prize money of 10000 dollars. Class methods, like class variables, can be accessed using the class itself. So you can call `Competition.get_raise_amount`. [Video description begins] *She pastes the following code in the subsequent code cell: `Competition.get_raise_amount()`.* [Video description ends]

`Competition` is the name of the class, not an object of that class. [Video description begins] *She points to the word `Competition` in the first line of code in the `Competition` class definition.* [Video description ends] And this will return 1.04. [Video description begins] *She points to the number in the second line of code in the `Competition` class definition.* [Video description ends]

Now class methods can be accessed using an object of the class as well. [Video description begins] *She pastes the following code in the subsequent code cell: `sprint.get_raise_amount()`. Also, she highlights the tenth, eleventh, and twelfth lines of code in the `Competition` class definition.* [Video description ends]

You can see that this is very similar to how class variables work. [Video description begins] *She runs the following code: `sprint.get_raise_amount()`.* [Video description ends] This will give you the same answer, `get_raise_amount` is 1.04. I'm now going to use an object to invoke the class method `set_raise_amount`. And I'm going to change the raise amount to 1.06. It was earlier 1.04. [Video description begins] *She pastes the following code in the subsequent code cell: `sprint.set_raise_amount(1.06)`.* [Video description ends]

Let's invoke the `get_raise_amount` class method using our `Sprint` object, and this gives us the new value of the `__raise_amount`, 1.06. [Video description begins] *She pastes the following code in the subsequent code cell: `sprint.get_raise_amount()`.* [Video description ends] As we've seen, class methods can be invoked using the class itself as well as an object instance. [Video description begins] *She pastes the following code in the subsequent code cell: `Competition.get_raise_amount()`.* [Video description ends]

Here I've used the `Competition` class to invoke the `get_raise_amount` class method, and this gives me the same return value; 1.06 is the new `__raise_amount`. Class methods are often useful as builder functions, functions which return an instance of a class by parsing some data. [Video description begins] *She pastes the following code in the subsequent code cell: `swimming_str = 'Swimming-8000'`.* [Video description ends]

Let's see what I mean in just a bit. Here is a swimming string, which contains the swimming competition and the prize money for swimming which is 8000 dollars. This is just a hyphenated string. I'll need to do a little parsing if I want the name of the competition and the prize money separately. And this parsing is simply splitting the string on the hyphen or the dash.

Splitting on the dash gives us a name of the competition and the prize money, which I store in the `name` and `prize` variables respectively. [Video description begins] *She pastes the following code in the subsequent code cell: `name, prize = swimming_str.split('-')`. She points to the dash in the code.* [Video description ends]

Now let's instantiate a `Competition` object. This is the swimming competition. Pass in the name and the prize money that we parsed from our swimming string. [Video description begins] *She*

pastes the following code in the subsequent code cell: swimming = Competition(name, prize).
[Video description ends]

If you now invoke the print_details instance method on the swimming object, it'll print out the details of this competition. The Name of the competition is Swimming and the prize money is 8000 dollars. [Video description begins] *She pastes the following code in the subsequent code cell: swimming.print_details(). She then runs the code. Also, she highlights the eighth and the ninth lines of code in the Competition class definition.* [Video description ends]

Now that we've seen how we can parse details from a string and instantiate an object of the type Competition, let's see this in action using a class method. Here is the same class Competition that we've seen earlier with a few additional methods.

[Video description begins] *She pastes a code comprising nineteen lines in the subsequent code cell. The following is the code per line: Line 1: class Competition:, line 2: __raise_amount = 1.04, line 3: def __init__(self, name, prize):, line 4: self.__name = name, line 5: self.__prize = prize, line 6: def raise_prize(self):, line 7: self.__prize = self.__prize * self.__raise_amount, line 8: def print_details(self):, line 9: print("Name: {}, prize: {}".format(self.__name, self.__prize)), line 10: @classmethod, line 11: def get_raise_amount(cls):, line 12: return cls.__raise_amount, line 13: @classmethod, line 14: def set_raise_amount(cls, amount):, line 15: cls.__raise_amount = amount, line 16: @classmethod, line 17: def from_str(cls, competition_str):, line 18: name, prize = competition_str.split('-'), and line 19: return cls(name, prize).* [Video description ends]

You can see we have the same class variable __raise_amount, we have the init, raise_prize, and print_details methods. These are instance methods. Then we have three class methods after that. The two class methods we've seen before, get_raise_amount and set_raise_amount, I've added a third class method here called from string. [Video description begins] *After highlighting the last four lines of code, she points to the following code fragment in the seventeenth line: from_str.* [Video description ends]

The from_str method returns an instance of a competition class by parsing the string that we pass in. This creates a competition object from an input string. The input arguments to this from_str class method is the reference to the class itself, that is the first cls variable. And the second input argument is the competition string, which will specify the name and the prize money separated by a hyphen.

Within this from string class method, we will extract the name and the prize money by splitting on the hyphen and create an instance of the Competition class. Observe how we create an instance of the Competition class by invoking the cls variable. [Video description begins] *She highlights the last two lines of code.* [Video description ends]

This bears out that the first argument in a class method is a reference to the class itself. [Video description begins] *She points to the following code fragment in the last line of code: cls.* [Video description ends] And we use this reference to the class to create an object of that class. [Video description begins] *After pointing to the text cls in the seventeenth line of code, she points to the same text in the last line of code.* [Video description ends]

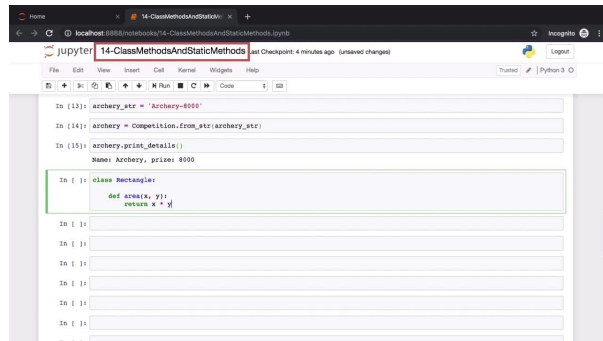
We are now ready to see this from string class method in action. I'm going to set up a string here called Archery-8000. Archery is the name of the competition, 8000 dollars is the prize money. [Video description begins] *She pastes the following code in the subsequent code cell: archery_str = 'Archery-8000'.* [Video description ends]

Now I'm going to pass this archery string into Competition.from_str; from_str will parse this archery string and return a competition object for the type archery. [Video description begins] *She pastes the following code in the subsequent code cell: archery = Competition.from_str(archery_str). She also highlights the sixteenth, seventeenth, and eighteenth lines of code in the Competition class definition.* [Video description ends]

And if you print out the details of this archery object, you can see that the competition Name is

Archery, prize money is 8000 dollars. The example that we just saw here is a pretty standard use case for a class method in Python. [Video description begins] *She pastes the following code in the subsequent code cell: archery.print_details(). And after running the code, she points to __name and __prize in the ninth line of code in the Competition class definition.* [Video description ends]

12. Video: Static Methods (it_pycipydj_04_enus_12)



```
In [13]: archery_str = "Archery-8000"
In [14]: archery = Competition.from_str(archery_str)
In [15]: archery.print_details()
Name: Archery, prize: 8000

In [ ]: class Rectangle:
    def area(x, y):
        return x * y

In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
```

In this video, you will learn how to work with utility methods in classes.

- *work with utility methods on classes*

[Video description begins] *Topic title: Static Methods. Your host for this session is Janani Ravi.* [Video description ends]

Now that we've understood what class methods are, let's move on to another kind of method that you might define on a class. This is a static method. Now class methods have a reference to the class itself. Static methods have nothing, no reference to the class, nothing. These are simply utility methods, which make sense as a part of a class but have nothing to do with the class itself.

Static methods are class methods in that they are defined within a class but they are utility methods. Static methods cannot access the state associated with the class. That means, if you have class variables within a class, static methods cannot access those. They also cannot access the state of instance variables or instance state.

[Video description begins] *The Jupyter Notebook is open. It includes a workspace comprising several code cells. A code comprising three lines displays in one of the code cells. The following is the code per line: Line 1: class Rectangle:, line 2: def area(x, y):, and line 3: return x * y.* [Video description ends]

Here is an example of a method which is just a regular method within a class. I haven't converted it to a static method yet. This is a class Rectangle. And there is a method within this class which helps us calculate the area of a rectangle. [Video description begins] *The presenter highlights the second and the third line of code.* [Video description ends]

The input arguments to this area method are x and y, the length and breadth of a rectangle. And the area is simply x multiplied by y. [Video description begins] *She highlights the third line of code.* [Video description ends] Now I'm going to convert this area method to a static method in a very manual way; staticmethod is a built-in function available in Python that allows you to convert a method to a static method. [Video description begins] *She pastes the following code in the subsequent code cell: Rectangle.area = staticmethod(Rectangle.area).* [Video description ends]

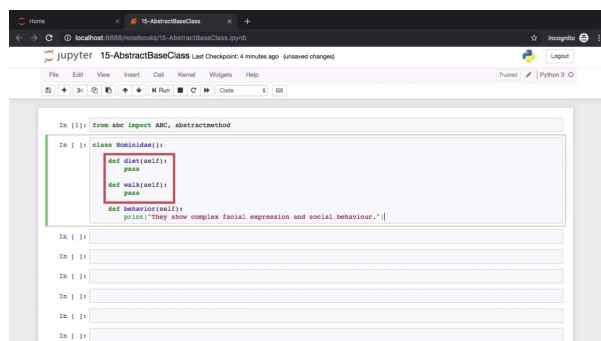
So I pass in the original `Rectangle.area` method and assign the new static method to `Rectangle.area` once again. Superficially, you will feel that static method seems similar to class method in that they are invoked using the name of the class. [Video description begins] *She pastes the following code in the subsequent code cell: `print('area of the rectangle is :', Rectangle.area(15, 16))`.* [Video description ends]

Observe how we invoke the area method here, `Rectangle.area`, and we pass in the length and the breadth of the rectangle. This method calculates the area of the rectangle and returns it; 240 is the area. The difference is that static methods cannot access class variables. Now in our earlier example, we converted a method to a static method in a very manual way.

If you were to actually define a static method within your class, you would use the `@staticmethod` decorator as you see here on screen. [Video description begins] *She pastes a code comprising four lines in the subsequent code cell. The following is the code per line: Line 1: `class Rectangle;` line 2: `@staticmethod`, line 3: `def area(x, y);` and line 4: `return x * y.`* [Video description ends]

When you decorate a function with the `@staticmethod` decorator, the function is passed into the static method function that we saw earlier – to convert a method to a static method. And that's it. Once you've specified the decorator, you can invoke the static method as we did before. Pass in the length and the breadth, and this will give you the area of our rectangle. [Video description begins] *She pastes the following code in the subsequent code cell and runs the same: `print('area of the rectangle is :', Rectangle.area(15, 16))`.* [Video description ends]

13. Video: Abstract Base Classes (it_pycipydj_04_enus_13)



How to define classes as abstract.

- *define classes as abstract*

[Video description begins] *Topic title: Abstract Base Classes. Your host for this session is Janani Ravi.* [Video description ends]

We worked with classes and inheritance before, we know what base classes are, and we know what derived classes are. Now all of the classes that we've seen so far can be instantiated. We can create objects of all of the classes whether they are base classes or derived classes. Now this may not be something that you want.

You might have a base class which is not instantiable, which you can't create objects of. That's when you'd have the base class be an abstract base class, or ABC. Abstract base classes and a way to tag abstract methods can be imported from the ABC module. Go ahead and set up the import statement as you see here on screen. [Video description begins] *The Jupyter Notebook is open. It includes a workspace comprising several code cells. The following code displays in the first code cell: `from abc import ABC, abstractmethod.`* [Video description ends]

We'll talk about how we can use this ABC base class and the abstract method in order to make classes that are not instantiable. Here is a class which is a base class for me. The base class is called Hominidae, and it inherits from the default object base class.

[Video description begins] *The presenter pastes a code comprising seven lines in the subsequent code cell. The following is the code per line: Line 1: class Hominidae(); line 2: def diet(self); line 3: pass, line 4: def walk(self); line 5: pass, line 6: def behavior(self); and line 7: print("They show complex facial expression and social behavior.").* [Video description ends]

Now there are three functions here. Two of the functions do not have any implementations. We have pass within the function body. One function, the behavior function, does have an implementation. The two functions that don't have implementations, diet and walk, we simply say pass indicating that we have no diet or how they walk specified for this base class Hominidae.

But we know that the behavior of an Hominidae is that they show complex facial expression and social behavior. [Video description begins] *She highlights the last two lines of code.* [Video description ends] Even though we don't have implementations for the diet and walk methods of this class, you can instantiate objects of the Hominidae class. [Video description begins] *She pastes a code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: chimpanzee = Hominidae() and line 2: chimpanzee.behavior().* [Video description ends]

I've created an instance of the Hominidae class here and stored it in the chimpanzee variable. I invoke the behavior function on the chimpanzee object. And it prints out to screen: They show complex facial expression and social behavior. Now it's possible for you to invoke the diet function as well as the walk function in this chimpanzee object and you'll get no output. [Video description begins] *She pastes the following code in the subsequent code cell: chimpanzee.diet(). She pastes the following code in the subsequent code cell: chimpanzee.walk().* [Video description ends]

Because we have pass within both of these function bodies, the pass statement simply does nothing, no code is executed. [Video description begins] *She points to the third line and the fifth line of code in the Hominidae class definition.* [Video description ends]

Now I'm going to set up a derived class called Human, which derives from the base Hominidae class. [Video description begins] *She pastes a code comprising five lines in the subsequent code cell. The following is the code per line: Line 1: class Human(Hominidae); line 2: def diet(self); line 3: print("Humans are omnivorous."); line 4: def walk(self); and line 5: print("They are bipeds.").* [Video description ends]

And within this Human class, I'm going to define the function diet and walk. Both of these functions will now have implementations. The implementations here are very simple, they're just print statements. The diet says, Humans are omnivorous, and the walk functions says, They are bipeds. If you instantiate an object of the class Human, all of the functions from the base Hominidae class now have implementation.

[Video description begins] *She pastes a code comprising three lines in the subsequent code cell. The following is the code per line: paul = Human(), line 2: paul.diet(), and line 3: paul.walk().* [Video description ends]

The variable paul holds an object of type Human. I invoke paul.diet and paul.walk, and the corresponding print statements are printed out to screen. Now if you think about it, it actually makes no sense to be able to instantiate objects of the class Hominidae. Because the base class is just partially defined, it doesn't define a species completely. The derived class Human should be instantiable.

The base class Hominidae should not be, which means you need to specify that Hominidae is a abstract base class. [Video description begins] *She points to the word Human and then to the*

word Hominidae in the first line of code in the Human class definition. [Video description ends]

And the way you do it in Python is by having the Hominidae base class inherit from ABC. [Video description begins] *She pastes a code comprising seven lines in the subsequent code cell. The following is the code per line: Line 1: class Hominidae(ABC);, line 2: def diet(self);, line 3: pass, line 4: def walk(self);, line 5: pass, line 6: def behavior(self);, and line 7: print("They show complex facial expression and social behavior.").* [Video description ends]

ABC is an acronym for abstract base class, and it's a class available within the ABC namespace in Python. [Video description begins] *She points to the following code fragment in the first line of code in the Hominidae class definition: ABC.* [Video description ends]

The term abstract essentially means that the class is incomplete, it does not have implementations for all methods. So the methods diet and walk do not have implementations in the Hominidae class. It's an abstract base class. When you inherit a class from ABC, it's an indication to whoever is using your code that this is an abstract base class, it's an incomplete class.

[Video description begins] *She pastes a code comprising two lines in the subsequent code cell. The following is the code per line: Line 1: chimpanzee = Hominidae() and line 2: chimpanzee.behavior().* [Video description ends]

However, the behavior of the class does not change. You can still create instances of the Hominidae class. If you invoke the behavior function, you'll get the expected result. [Video description begins] *The following output displays below the code cell: They show complex facial expression and social behavior.* [Video description ends]

And if you invoke the diet and walk functions, you'll find that there is no implementation, it returns nothing. [Video description begins] *She points to the portion of the code starting at line 2 and ending at line 5 in the Hominidae class definition.* [Video description ends]

Once again, let's redefine the Human class, which derives from Hominidae and has implementations for both the diet as well as walk functions. [Video description begins] *She pastes a code comprising five lines in the subsequent code cell. The following is the code per line: Line 1: class Human(Hominidae);, line 2: def diet(self);, line 3: print("Humans are omnivorous."), line 4: def walk(self);, and line 5: print("They are bipeds.").* [Video description ends]

You can instantiate objects of the Human class as well, invoke diet, and walk, and you'll get the expected results. [Video description begins] *She pastes a code comprising three lines in the subsequent code cell. The following is the code per line: Line 1: myra = Human(), line 2: myra.diet(), and line 3: myra.walk(). And when she runs the code, an output comprising the following two lines displays below the code cell: Line 1: Humans are omnivorous. and line 2: They are bipeds.* [Video description ends]

As we've mentioned before, ABC is a class available in the abc namespace in Python. [Video description begins] *She pastes the following code in the subsequent code cell: help(ABC).* [Video description ends]

The namespace abc is all in lowercase. Passing this ABC class to the help function will give you additional details of this class. Observe that you can specify abstract methods in your abstract base class. Let's see how.

[Video description begins] *She pastes a code comprising eight lines in the subsequent code cell. The following is the code per line: Line 1: class Hominidae(ABC);, line 2: @abstractmethod, line 3: def diet(self);, line 4: pass, line 5: def walk(self);, line 6: pass, line 7: def behavior(self);, and line 8: print("They show complex facial expression and social behavior.").* [Video description ends]

What you really want to do in the real world is to enforce that a class is abstract and should not be instantiated. And the way you do it is inherit from the ABC base class, as we do here, class Hominidae inherits from ABC. [Video description begins] *She points to the following code fragment in the first line of code: ABC.* [Video description ends]

And for those functions in this Hominidae base class, for which you do not have implementations, you decorate it using the @abstractmethod decorator. [Video description begins] *She highlights the second, third, and fourth lines of code.* [Video description ends]

The presence of this @abstractmethod decorator on the diet function will ensure that no instances of the Hominidae class can be instantiated. I haven't decorated the walk function yet, and I'll do that in just a bit. [Video description begins] *She highlights the fifth and the sixth lines of code.* [Video description ends]

Now with this setup, let's try and create an instance of the Hominidae class. [Video description begins] *She pastes the following code in the subsequent code cell: great_apes = Hominidae().* [Video description ends]

And I'm going to assign it to the great_apes variable. When you execute this code to create an instance of the Hominidae class, you'll encounter an error. The TypeError very clearly says, Can't instantiate abstract class Hominidae with abstract methods diet. It very clearly indicates that the diet method is abstract, it does not have an implementation, you cannot instantiate the Hominidae class. [Video description begins] *She highlights the second, third, and fourth lines of code.* [Video description ends]

Now let's set up the Human class as before, which derives from the Hominidae base class, and let's add implementations for the diet and walk methods. If all of the abstract methods in the base class have concrete implementations in the derived class, you'll be able to instantiate objects of the class Human. Here bill is a Human. You can invoke the diet and walk functions on the bill object and they will all work well.

[Video description begins] *She pastes and runs a code comprising the following three lines in the subsequent code cell: Line 1: bill = Human(), line 2: bill.diet(), and line 3: bill.walk().* [Video description ends]

Let's extend our example. [Video description begins] *She pastes a code comprising seven lines in the subsequent code cell. The following is the code per line: Line 1: class Hominidae(ABC), line 2: @abstractmethod, line 3: def diet(self);, line 4: pass, line 5: @abstractmethod, line 6: def walk(self);, and line 7: pass.* [Video description ends]

Here is the class Hominidae once again. It inherits from the ABC base class, indicating that it's an abstract base class. Now both of our methods, diet as well as walk, are both decorated using @abstractmethod. This decorator ensures that a class can be instantiated only when it has concrete implementations for these abstract methods.

Remember, concrete implementations need to be present for all abstract methods before the class can be instantiated. [Video description begins] *A banner comprising the following two lines of text displays at the bottom: Line 1: All abstract methods should have implementations and line 2: Otherwise Python will not allow instances of the class to be created.* [Video description ends]

This will become clearer when we look at an example. Consider the class Human here. [Video description begins] *She pastes a code comprising three lines in the subsequent code cell. The following is the code per line: Line 1: class Human(Hominidae);, line 2: def diet(self);, and line 3: print("Humans are omnivorous").* [Video description ends]

We only have an implementation for the diet method, which was abstract in the base class. [Video description begins] *She highlights the second and the third line of code.* [Video description ends] We don't have an implementation for the walk method. [Video description

begins] *She points to the sixth line of code in the Hominidae class definition.* [Video description ends]

Now if you try to instantiate an object of the class Human, you'll encounter an error. The TypeError says, Can't instantiate abstract class Human with abstract method walk. [Video description begins] *She pastes and runs the following code in the subsequent code cell: cathy = Human().* [Video description ends]

You can only instantiate derived classes that have implementations for both diet as well as walk, as per the class that you see here on screen. [Video description begins] *She pastes a code comprising five lines in the subsequent code cell. The following is the code per line: Line 1: class Human(Hominidae); line 2: def diet(self); line 3: print("Humans are omnivorous."); line 4: def walk(self); and line 5: print("They are bipeds.");* [Video description ends]

We provided concrete implementations for both methods marked abstract in the base class. We can now instantiate objects of type Human. And you can see that this instantiation and the invocation of the diet and walk functions was successful.

[Video description begins] *After highlighting the portion of the code starting at the second line and ending at the fifth line, she pastes and runs a code comprising three lines in the subsequent code cell. The following is the code per line: Line 1: cathy = Human(), line 2: cathy.diet(), and line 3: cathy.walk().* [Video description ends]

14. Video: Exercise: Advanced Functionality in Classes (it_pycipydj_04_enus_14)



In this exercise, you will

- List some of the special methods that can be specified in Python classes and what they represent
- Define a Python class, Employee, and create a property within it called employee_id using decorators in Python
- Differentiate between class methods and static methods in Python

After completing this video, you will be able to list special methods and what they represent, define a class and create a property within it, and differentiate between class methods and static methods.

- *list special methods and what they represent, define a class and create a property within it, and differentiate between class methods and static methods*

[Video description begins] *Topic title: Exercise: Advanced Functionality in Classes. Your host for this session is Janani Ravi.* [Video description ends]

Now that you're familiar with some of the advanced functionality associated with Python classes, let's perform a few exercises. In this exercise, you will list some of the special methods that can be specified in Python classes and what these special methods represent. You will then move on to writing some code, try and define a Python class named Employee, and create a property within it called employee_id using decorators in Python.

And finally, you will differentiate between class methods and static methods in Python. Now would be a good time for you to pause this video and try and answer these questions on your own. We'll then move on to looking at a few sample answers. Remember that your answers

might be a little different, and that's totally fine as long as the fundamental concepts mentioned are exactly the same.

[Video description begins] *A solution banner appears at the bottom.* [Video description ends]

The special method that is used most widely in Python classes is the `__init__` special method that we were introduced to earlier; `__init__` is what we use to initialize the member variables of the class to set up the initial state of a class. [Video description begins] *On the slide, the `__init__` method is written as follows: `__init__()`.* [Video description ends]

The `__init__` method is automatically invoked when you instantiate objects of a Python class. One of the most common functions used in Python is the built-in `len` function. Now the built-in `len` function, you know works with lists and tuples. But these are built-in classes in Python. How do you have the `len` function work with your custom data types, your custom classes? Well, you'll implement the `len` special method.

Implementing the `len` special method allows us to use objects of your custom class as input arguments to the built-in `len` function. [Video description begins] *On the slide, the `len` special method is written as follows: `__len__()`.* [Video description ends]

Python allows you to use special methods so that objects of your custom class can be used as operands for arithmetic operators such as addition, subtraction, multiplication, division, and so on. For example, the `__add__` special method allows you to overload the plus operator and this will allow you to add two objects of your custom class. [Video description begins] *On the slide, the `__add__` special method is written as follows: `__add__()`.* [Video description ends]

For loops and while loops are extremely useful programming constructs. They work with lists, dictionaries, tuples. You can also have these work with your custom classes if you implement an `__iter__` special method. The `__iter__` special method returns an iterator which allows users to iterate over the elements of your custom class using for loops or even while loops; `__iter__` is typically used along with the `__next__` special method, which actually implements the iterator and gives you the next element in sequence. [Video description begins] *On the slide, the `__iter__` special method is written as follows: `__iter__()`.* [Video description ends]

Now let's move on to answering the next question, where we'll set up an `Employee` class and specify properties within it using decorators. [Video description begins] *On the slide, the `__next__` special method is written as follows: `__next__()`.* [Video description ends]

Here is the class `Employee`, which inherits from the object base class. The `__init__` method specifies just one instance variable, that is, the `employee_id`.

[Video description begins] *A code comprising twelve lines displays on a terminal. The following is the code per line: Line 1: `class Employee(object):`; line 2: `def __init__(self, employee_id):`; line 3: `self.__employee_id = employee_id`, line 4: `@property`, line 5: `def employee_id(self):`, line 6: `return self.__employee_id`, line 7: `@employee_id.setter`, line 8: `def employee_id(self, employee_id):`, line 9: `self.__employee_id = employee_id`, line 10: `@employee_id.deleter`, line 11: `def employee_id(self):`; and line 12: `del self.__employee_id`.* [Video description ends]

Now this instance variable is what I'm going to convert to a property using the `@property` decorator on the `employee_id` function. Now the name of the function that you specify here is important because this specifies the name of the property. Now the setter for this property is decorated using the `@employee_id.setter` decorator. These indicates that this is a setter function for the `employee_id` property, and here is an implementation of the setter function.

Observe that the setter takes in a new value of `employee_id`, which is used to update the current value of `employee_id`. And finally, we have the deleter function here for the `employee_id`, `employee_id.deleter`, which simply calls `del` on the `employee_id` instance variable. And finally, we'll move on to the last question here in this exercise, the difference between class methods and static methods in Python.

A class method in Python is used to modify the state of the class itself, that is, access class variables and maybe update class variables. A static method just happens to be associated with a particular class. It's just a utility method. It cannot access or modify class state. When you define a class method in Python, the first input argument to a class method is a reference to the class itself, typically represented using the `cls` variable.

Static methods do not contain a reference to the class itself, which is why they cannot update class state. Static methods are general utility methods which are loosely associated with the class, but they can be standalone methods if you want to. They're associated with the class because there is a logical connection with the class. And the last difference here is the way they're defined. We'll use the `@classmethod` decorator to specify a class method whereas static methods are specified using the `@staticmethod` decorator.

Course File-based Resources

- [Course 4 Assets](#)

Assets