Complex Data Types in Python: Working with Lists & Tuples in Python

Learn how to work with lists, tuples, and strings in Jupyter notebook in Python in this 14-video course. You will discover similarities and differences between tuples and lists and see how strings are essentially just a list of characters. Begin with an introduction to lists, and then create and initialize lists in Python. You will then access and update list elements; add, remove, sort, and reverse elements from a list; execute built-in functions with lists, and create new lists from existing lists by using slicing operations. Next, examine how to extract specific elements from the original list using step size; perform list functions on strings; invoke functions on the string object; and access substrings with slicing operations. Receive an introduction to tuples, exploring the similarities between lists and tuples, then move on to understanding tuple immutability by specifying differences between lists and tuples. Then an introduction to other complex data types and using dictionaries and sets in Python. The concluding exercise concerns recalling differences and similarities between lists and tuples.

Table of Contents

1. Course Overview
2. Introducing Lists
3. Performing Simple List Operations
4. Performing Useful List Operations
5. Using Built-in Functions with Lists
6. Perform Slicing Operations on Lists
7. Using Step Size in Slicing Operations
8. Working with Strings as a List of Characters
9. Invoking Functions on Strings
10. Perform Slicing Operations on Strings
11. Introducing Tuples
12. Understanding Tuple Immutability
13. Introducing Other Complex Data Types
14. Exercise: Lists, Tuples, Similar Yet Different

Course Overview

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

Hello there, welcome to this course working with lists and tuples in Python. My name is Janani Ravi, and I will be your instructor for this course.

[Video description begins] *Janani Ravi is a Software engineer and big data expert*. [Video description ends]

A little about myself first, I'm a co-founder 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. Python is fast becoming more than a programming language, it's now the gateway into the world of big data, machine learning, and artificial intelligence.

Simple, transparent and easy to use, Python is ubiquitous these days. In this course, you will continue your exploration of fundamental building blocks of Python, specifically working with list, tuples, and with strings viewed as a list of characters. This course assumes that you have an interactive Python development environment such as Jupyter installed, and are familiar with defining variables and invoking simple built-in functions. In this course, you will work with the list, an extremely important data type in Python.

You will see how lists are initialized, and how values can be inserted into lists, and how indexing can be used to find values within lists. You'll also see how strings are effectively just lists of characters. Then you'll move on to list slicing, a technique used to access different parts of a list possibly all at once using concise syntax. List slicing is basically a way to create a new list with some selected portions of the data in the original list.

It plays an important role in pre-processing data in applications, such as big data and machine learning. So a strong grasp of these fundamentals will serve you extremely well. You'll then move on to tuples, which are great for representing coordinate data, such as x and y coordinates. By the end of this course, you will be able to create lists and tuples insert data into them, and access and manipulate that data. You'll know how to use list slicing as well as tuple slicing to effectively select parts of a list, and how to use list manipulation operations to work with strings which are just lists of characters.

Introducing Lists

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

In this video, we'll get more closely introduced to a complex data type that we'll use often in Python that is the Python list.

[Video description begins] *A jupyter notebook titled IntroducingPythonLists displays in the browser window. The page URL is localhost:8888/notebooks/IntroducingPythonLists%20ipynb. The notebook has a menu bar and a tool bar. The menu bar contains the following options: File, Edit, Insert, Cell, and so on. The tool bar contains the icons for the following options: Save and Checkpoint, insert cell below, Run, and so on. Below the tool bar is the main notebook area. It contains many empty code cells with the label In [ ] to their left.* [Video description ends]

A list in Python is just an ordered collection of elements, and these elements are specified within square brackets. So an empty_list in Python is simply two square brackets with nothing in between.

[Video description begins] *The statement: empty_list = [ ] displays in the first code cell and is highlighted. There are two arrows pointing to the square brackets.* [Video description ends]

The empty_list variable is a list with no elements within it.

[Video description begins] *The second code cell displays the statement: empty_list, which is highlighted.* [Video description ends]

And when you type out empty_list, you'll see in your Jupyter notebook that there's nothing. Only the square brackets representing that it's a list.

[Video description begins] *The code cells now show numbering 1 and 2 for the first two code cells respectively. The numbering appears inside the In square brackets. The output [ ] displays below the second code cell. It is labeled: Out [2]:.* [Video description ends]

A list is an ordered collection, and it can contain a collection of any kind of elements. So the elements within a list can be of any data type.

[Video description begins] *The third code cell displays the following highlighted statement: list_str = ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla']. It is followed by the second statement: list_str.* [Video description ends]

Here you see a list of strings, where the strings are names of cars. Observe that the list as a whole is enclosed in square brackets, that's how we know it's a list. And you can see that individual elements are separated from one another using commas. The variable list_str is a list of strings as displayed here on screen.

[Video description begins] *She points to the open and close square brackets of the highlighted statement. She then points to the commas between Toyota Camry, Honda Accord, Honda Civic, and Toyota Corolla. She runs the cell and the output displays the label: Out [3]: followed by the output ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla'].* [Video description ends]

The list is a special complex data type supported in Python. The type of this variable, list_str, is simply list.

[Video description begins] *The fourth code cell displays the following highlighted statement: type(list_str). The following output label and output display below it: Out [4]: list.* [Video description ends]

The elements within a list can be of any data type. Here is a list of integers where all of the elements within the list_int variable are integers.

[Video description begins] *The fifth code cell displays the following highlighted statement: list_int = [5 ,7, 8 ,9,20,40]. It is followed by the second statement: list_int. The following output label and output display below it: Out [5]: [5,7, 8 ,9,20,40].* [Video description ends]

The square brackets and the comma separating the elements tells us that this is a list. Let's see a list of floating point values as well. You can have elements within a list be floats as in the case of list_float.

[Video description begins] *The sixth code cell displays the following highlighted statement: list_float = [5.5 ,7.8, 8.5 ,9.2,20.3,40.6]. It is followed by the second statement: list_float.* [Video description ends]

Let's go on, you can have a list of Boolean values as well.

[Video description begins] *The following output label and output display below the sixth code cell: Out [6]: [5.5 ,7.8, 8.5 ,9.2,20.3,40.6].* [Video description ends]

Remember Boolean values are True, False values. And these Boolean values can be the elements in a list.

[Video description begins] *The seventh code cell displays the following highlighted statement: list_bool = [True ,False]. It is followed by the second statement: list_bool.* [Video description ends]

There are some details you need to remember when you're working with list.

[Video description begins] *The following output label and output display below the seventh code cell: Out [7]: [True,False].* [Video description ends]

List elements can be duplicates, so lists can contain duplicate values, and that's totally fine. Also, the main feature of lists is the fact that the ordering of elements in a list is significant. The order in which the elements appear in the list is the order in which the elements can be accessed when you iterate over a list. Python lists can contain elements of data types that are different from one another. Here is an example of a mixed_list. It contains a string, an integer, a floating point value, and a Boolean value. Elements of different data types can belong to the same list.

[Video description begins] *The eighth code cell displays the following highlighted statement: mixed_list = ['Nissan Santra', 1,2.5,True]. It is followed by the second statement: mixed_list. The following output label and output display below the eighth code cell: Out [8]: ['Nissan Santra' , 1,2.5,True].* [Video description ends]

Let's see how we can access individual elements in a list. Here we have a cars_list with a list of cars, which are just string values.

[Video description begins] *The ninth code cell displays the following highlighted statement: cars_list = ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla']. It is followed by the second statement: cars_list. The following output label and output display below the ninth code cell: Out [9]: ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla'].* [Video description ends]

Individual elements in a list can be accessed using indexes or index positions. And these indexes are specified within square brackets.

[Video description begins] *The tenth code cell displays the following highlighted statement: print(cars_list [0]).* [Video description ends]

Another detail to note here is that the first element in the list is at index position 0.

[Video description begins] *She points to index 0 between the square brackets.* [Video description ends]

So list indices in Python start from 0. So if you want to access the first element in a list, within square brackets, you need to specify the integer 0, and that's what we've done here.

[Video description begins] *The screen shows a highlight around the square brackets in the cars list. The list elements, Toyota Camry, Honda Accord, Honda Civic, and Toyota Corolla, are graphically labeled 0, 1, 2, and 3 respectively.* [Video description ends]

And when you call this print statement, you'll see that we access and print out the first element in the list which is the Toyota Camry.

[Video description begins] *She refers to the output of the tenth code cell, which is the Toyota Camry.* [Video description ends]

The second element in the list will be at index position 1. The third element at index position 2.

[Video description begins] *The eleventh code cell displays the following statements. Line 1: print(cars_list [1]). Line 2: print(cars_list [2]). Line 3: print(cars_list [3]). The screen displays the cars_list statement towards the center-right and the elements, Toyota Camry, Honda Accord, Honda Civic, and Toyota Corolla, are graphically labeled 0, 1, 2, and 3 respectively.* [Video description ends]

And the fourth element at index position 3. And these are what we're going to print out here.

[Video description begins] *The output is as follows. Line 1: Honda Accord. Line 2: Honda Civic. Line 3: Toyota Corolla.* [Video description ends]

These are the Honda Accord, Honda Civic, and the Toyota Corolla. The first element of the list is at index position 0. It stands to reason that the last element of the list will be at index position length -1. So if you have four elements in a list, index position 4 is actually not valid. So the last element will be at 4-1 index position 3. And if you try access the element at position 4, well, that's an error.

[Video description begins] *She points to the following statement on the twelfth code cell: print(cars_list [4]). The screen displays the following error message: IndexError: list index out of range.* [Video description ends]

You can see that the Python error here is quite descriptive. It clearly says, list index out of range. For the four element cars_list, there is no element at index position 4. The last element is at index position 3.

[Video description begins] *The screen displays the red highlighted cars_list statement towards the center. She highlights the last element, Toyota Corolla in yellow from the cars_list elements.* [Video description ends]

If you remember the len built-in function in Python, previously we had used this len function with strings. However, the len function also works with lists.

[Video description begins] *The thirteenth code cell displays the following statement: len(cars_list).* [Video description ends]

And when you pass in a list to the len function, it'll print out the number of elements in the list, which in our case here is 4. Let's work with our list of cars and using the print function once again.

[Video description begins] *The following output label and output display below it: Out [13]: 4.* [Video description ends]

The car at index position 2 access elements at specific indices using square brackets, we specified 2 within square brackets.

[Video description begins] *The fourteenth code cell displays the following statement: print("car at index 2: ", cars_list [2]). The screen also displays the red highlighted cars_list statement towards the center. She highlights the third element, Honda Civic, in yellow from the cars_list elements.* [Video description ends]

And that is the Honda Civic. Remember that this is the third element in our cars_list. Our cars_list has four cars in total, the length of the list was four.

[Video description begins] *The following output displays below: car at index 2: Honda Civic.* [Video description ends]

So you access the very last element by passing in the index value length minus 1, that is 3, and we can access the Toyota Corolla.

[Video description begins] *The fifteenth code cell displays the following statement: print(cars_list [3]). The output displays the Toyota Corolla.* [Video description ends]

The cars_list is an example of a simple list which is a single sequence of strings. This is often referred in mathematics as a vector.

[Video description begins] *The sixteenth code cell displays the following statement: print(cars_list [1, 2]).* [Video description ends]

A vector has just one dimension, and you use only one index to access the element within a vector. So if you try to access this list using two index values like this, that is an error.

[Video description begins] *She highlights the index values 1, 2 in the statement and the error message, which reads as follows appears: TypeError: list indices must be integers or slices, not tuple.* [Video description ends]

Python supports very complex and powerful operations on list, allowing you to access specific positions or elements within a list.

[Video description begins] *The seventeenth code cell displays the following highlighted statement: cars_list. The following output label and output display below the code cell: Out [17]: ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla'].* [Video description ends]

Here is our list of cars, you'll see that Python also supports negative index values.

[Video description begins] *The eighteenth code cell displays the following statement: print(cars_list [-1]). The screen also displays the red highlighted cars_list statement towards the center. She graphically labels the fourth element Toyota Corolla as -1.* [Video description ends]

We're using the square brackets to index into our cars_list, but we've specified -1 as the index position. This tells Python that we want to read the list backwards, and we want to access the last element in the list. -1 references the last element in the list. You can see that Python has printed out the Toyota Corolla, which is the last element in our cars_list.

Negative values can be used to access the list from the last element onwards. So -2 is the index position of the second to last element in the list. -3 for the third to last element. -4 for the fourth to last element and so on.

[Video description begins] *The nineteenth code cell displays the following statements. Line 1: print(cars_list [-2]). Line 2: print(cars_list [-3]). Line 3: print(cars_list [-4]). The screen displays the cars_list statement towards center-right and the elements Toyota Camry, Honda Accord, and Honda Civic are graphically labeled -4, -3, and -2 respectively*. [Video description ends]

cars_list of -2 is the Honda Civic, second to last element, then we have the Honda Accord, and the Toyota Camry, which is the first element at index position -4. We have just four elements in this list. So if you try to access the element at index position -5, we'll find that Python rightfully will throw an error.

[Video description begins] *The twentieth code cell displays the following statement: print(cars_list [-5]). The screen also displays the red highlighted cars_list statement towards the center.* [Video description ends]

The list index is out of range. Negative indexes can be confusing when you first encounter them in Python. Just remember whenever you see the negative value, start from the last position in the list. So -2 is the second to last element in the list. And in our list, it happens to be the Honda Civic that is present at the second to last position.

[Video description begins] *She points to the twenty-first code cell with the following statement: cars_list [-2]. The screen also displays the red highlighted cars_list statement towards the center. The output displays 'Honda Civic'*. [Video description ends]

So we know that the cars_list has four elements, which we can get by calling the len function and passing in the cars_list as an input argument. Let's print out the cars_list once again so that we remind ourselves what the elements within this list are.

[Video description begins] *She refers to the twenty-second code cell with the following statement: len(cars_list). The output displays the value 4*. [Video description ends]

And let's move on to some new operations that allow us to update the elements that are present in a list.

[Video description begins] *She refers to the twenty-third code cell with the following statement: cars_list. The output shows the cars_list elements enclosed in the square brackets separated by commas*. [Video description ends]

Observe here that we are accessing the cars_list at index position 3, and we are updating the element at this index position to be the Nissan Sentra.

[Video description begins] *The twenty-fourth code cell displays the following statements. Line 1: cars_list [3] = 'Nissan Sentra'. Line 2: print("New car at index 3 is: ", cars_list [3]). Line 3: cars_list*. [Video description ends]

cars_list[3] is on the left side of the assignment operator, and that is what will be updated. And if you print out the new car at index 3, you'll see that it's the Nissan Sentra. And the Toyota Corolla that existed earlier is no longer part of our list.

[Video description begins] *Following line displays in the output: New car at index 3 is: Nissan Sentra. Below this line is: ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Nissan Sentra']*. [Video description ends]

Just like we can use the square brackets to access individual index positions in a list, we can use the square brackets to update those list elements.

[Video description begins] *The twenty-fifth code cell displays the following statements. Line 1: cars_list [2] = 'Hyundai Elantra'. Line 2: cars_list.* [Video description ends]

Here the car at index position 2, that is the third car in our list, is updated to the Hyundai Elantra.

[Video description begins] *The following output label and output display below the code cell: Out [25]: ['Toyota Camry', 'Honda Accord', 'Hyundai Elantra', 'Nissan Sentra'].* [Video description ends]

So if you print out the cars_list now, you'll see that insteadof the Honda Civic, we have the Hyundai Elantra at index position 2.

Performing Simple List Operations

[Video description begins] *Topic title: Performing Simple List Operations. Your host for this session is Janani Ravi.* [Video description ends]

In this video, we'll see some of the operations that we can perform using lists.

[Video description begins] *A jupyter notebook titled MethodsInList-01 displays in the browser window. The page URL is localhost:8888/notebooks/MethodsInList-01%20ipynb. The main notebook area contains many empty code cells with the label In [ ] to their left.* [Video description ends]

We'll work with the same list that we had set up earlier, the cars_list with four cars. This is a list of strings, contains the Toyota Camry, Honda Accord, Honda Civic, and the Toyota Corolla.

[Video description begins] *The first code cell displays the following highlighted statement: cars_list = ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla']. It is followed by the second statement: cars_list. The following output label and output display below the code cell: Out [1]: ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla'].* [Video description ends]

We've already seen how we can use list indices to update a specific value in the list.

[Video description begins] *The second code cell displays the following statements. Line 1: cars_list [4] = 'Hyundai Elantra'. Line 2: cars_list. Line 1 is highlighted.* [Video description ends]

But what if we want to add a new element? We know that the last element in the cars_list is that index position 3, that is the Toyota Corolla.

[Video description begins] *She highlights the last element, Toyota Corolla, in cars_list.* [Video description ends]

What if we try to assign the Hyundai Elantra to index position 4? Well, this is an error. Python considers the list to have four elements up to index position 3.

[Video description begins] *A cross icon displays next to the Hyundai Elantra on line 1 of the second code cell. The following error message displays below the code cell: IndexError: List assignment index out of range.* [Video description ends]

So when we try to use index position 4 in our assignment statement, we get the error, which says list assignment index out of range. So when you're using square brackets, you can only access existing index positions in a list. If you want to add a new element, you need to invoke specific methods on a list.

[Video description begins] *The third code cell displays the following statements. Line 1: cars_list.append ('Hyundai Elantra'). Line 2: cars_list. Both the lines are highlighted.* [Video description ends]

And in this case, to append an element to the very end of the list, you call the append method. Notice how we say cars_list.append.

[Video description begins] *She highlights the .append command.* [Video description ends]

The dot basically says that this action will be performed on the cars_list. And will append the Hyundai Elantra to this list. If you see the cars_list, now you'll see that it has one new element. The Hyundai Elantra has been added to the very end. Append adds new elements to the very end of the list.

[Video description begins] *The following output label and output display below the third code cell: Out [3]: ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla' , 'Hyundai Elantra'].* [Video description ends]

Append is a method or a function that is called on the list itself, and it modifies the list itself.

[Video description begins] *The fourth code cell displays the following statements. Line 1: cars_list.append ('Nissan Sentra' , 'Ford Fusion'). Line 2: cars_list.* [Video description ends]

The append method or function that we invoke on the list can add just one element at a time. So if you call append with two elements, you see that this is an error in Python. Append takes exactly one argument and we've tried to pass in two.

[Video description begins] *The screen displays a cross icon at the end of the append statement and the following error message: TypeError: append() takes exactly one argument (2 given).* [Video description ends]

It's important to remember that the append method invoke or called on a particular list, changes the list itself. If you see the length of the cars_list now, you'll see that it has five elements up to index position 4.

[Video description begins] *She refers to the fifth code cell statement: len(cars_list). The following output label and output display below it: Out [5]: 5. She highlights the third cell output that displays the car names and also highlights the number 5.* [Video description ends]

If you want to insert an element at a specific index position in the list, you can use the insert function on a list. Insert takes in an index position and a list element, and tries to insert that element into the list.

[Video description begins] *The third code cell displays the following statements. Line 1: cars_list.insert (6, 'Nissan Sentra'). Line 2: cars_list.* [Video description ends]

It will shift other elements to the right if needed. Here, we're inserting the Nissan Sentra at element position 6.

[Video description begins] *The screen displays the existing cars list towards the center right.* [Video description ends]

We have just five elements in our list at this point in time from index positions 0 through 4. Now index position 6 is beyond the last element of the list. But that is totally fine. The Nissan Sentra will be added to the very end of the list as you can see from the result here.

[Video description begins] *The following output label and output display below the sixth code cell: Out [6]: ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla' , 'Hyundai Elantra', 'Nissan Sentra'].* [Video description ends]

If you pass in the list to the len built-in function, you'll see that the len function correctly calculates that the length of this list is six. There are six elements in this list.

[Video description begins] *She refers to the seventh code cell statement: len(cars_list). The output is 6.* [Video description ends]

Now that we've inserted the Nissan Sentra at index position 6, let's insert the Nissan Altima at index position 5.

[Video description begins] *The 8th code cell displays the following statements. Line 1: cars_list.insert (5, 'Nissan Altima'). Line 2: cars_list.* [Video description ends]

You'll see that Python remembers that Nissan Sentra, even though it's at the very end of the list was inserted using index position 6. So when you insert using index position 5, the Nissan Altima is added in as the second to last element in the list before the Nissan Sentra.

[Video description begins] *The following output label and output display below the 8th code cell: Out [8]: ['Toyota Camry', 'Honda Accord', 'Honda Civic', 'Toyota Corolla' , 'Hyundai Elantra' , 'Nissan Altima' , 'Nissan Sentra']. There is a small graphical arrow pointing to the sixth element, Nissan Altima.* [Video description ends]

These are the small details using list insert with index position values, and it's important that you remember how they work and their nuances. If you want to add multiple elements to the end of your list, you will use the list extend function rather than the append function.

[Video description begins] *The 9th code cell displays the following statements. Line 1: cars_list.extend (['Kia Optima', 'Kia Soul']). Line 2: cars_list.* [Video description ends]

cars_list.extend takes as its input argument another list, and this list contains the Kia Optima and the Kia Soul. These two cars will be added to the end of the cars_list. So if you print out the cars_list now, you will see that the Kia Optima and the Kia Soul are present at the very end of the list.

[Video description begins] *She refers to the output of code cell 9, which displays the car names in the following order: Toyota Camry, Honda Accord, Honda Civic, Toyota Corolla, Hyundai Elantra, Nissan Altima, Nissan Sentra, Kia Optima, and Kia Soul. They are separated by commas and listed within the square brackets.* [Video description ends]

You can intuitively work with lists using arithmetic operations as well.

[Video description begins] *The tenth code cell displays the following highlighted statement: cars_list2 = ['Hennessey Venom GT', 'Bugatti Veyron', 'Koenigsegg Agera R']. It is followed by the second statement: cars_list2.* [Video description ends]

Let's create a new list called cars_list2, which contains three other cars, very expensive ones.

[Video description begins] *The following output label and output display below the tenth code cell: Out [10]: ['Hennessey Venom GT', 'Bugatti Veyron', 'Koenigsegg Agera R'].* [Video description ends]

So if you have two list that you want to combine together, you can simply use the arithmetic + operation. The complete_cars_list is a combination or concatenation of cars_list1 and cars_list2.

[Video description begins] *The eleventh code cell displays the following highlighted statement: complete_cars_list = cars_list + cars_list2. It is followed by the second statement: complete_cars_list. She highlights the plus sign.* [Video description ends]

To take a look at the complete cars_list, all of the cars that you expect are in there.

[Video description begins] *The screen displays the cars_list and cars_list2 statements towards center-right with a plus icon in between. The output of the 11th code cell displays the car names in the following order: Toyota Camry, Honda Accord, Honda Civic, Toyota Corolla, Hyundai Elantra, Nissan Altima, Nissan Sentra, Kia Optima, Kia Soul, Hennessey Venom GT, Bugatti Veyron, and Koenigsegg Agera R. They are separated by commas and listed within the square brackets.* [Video description ends]

Observe that the cars which are in the cars_list appear first, and then come the cars in cars_list2.

[Video description begins] *She highlights the car entries from cars_list in the output. She then highlights the car entries from cars_list2 in the output.* [Video description ends]

When you concatenate two lists using the + operator, you create a brand-new concatenated list.

[Video description begins] *The 12th code cell displays the following statement: cars_list. The output for this code cell displays the elements in cars_list. She highlights the outputs of the code cells 11 and 12. There are arrows pointing to complete_cars_list and cars_list respectively.* [Video description ends]

The original lists are not affected. So if you take a look at cars_list, it contains the cars that it did originally. Lists support another function or method called index, which allows you to access the index position for a particular element.

[Video description begins] *The 13th code cell displays the following statement: complete_cars_list.index('Honda Civic'). The output displays the value 2.* [Video description ends]

Here when we ask for the index of the Honda Civic in the complete_cars_list, this list tells us that it is at index position 2, that is the third element in the list.

[Video description begins] *The screen displays the highlighted complete_cars_list statement towards center-right. Within the list, Honda Civic is highlighted. There is an arrow pointing to the text, Honda Civic, in the Input code cell.* [Video description ends]

Now, the list element that you pass into the index function has to be an exact match to the actual element present in the list.

[Video description begins] *The 14th code cell displays the following statement: complete_cars_list.index('honda Civic'). The alphabet "h" for honda, is in the small case whereas in complete_cars_list, it was in uppercase. She points to the uppercase alphabet "H" in the previous code.* [Video description ends]

So let´s say you have Honda Civic with the capital H within your list. And you pass in Honda Civic with the h in smaller case, this will not work. Python will not find Honda Civic, where h is in lower case.

[Video description begins] *A cross icon displays next to honda Civic on the code cell. The following error message displays below the code cell: ValueError: 'honda Civic' is not in list.* [Video description ends]

And this will be an error. The complete_cars_list at this point in time contains the Hyundai Elantra at index position 4.

[Video description begins] *The 15th code cell displays the following statement: complete_cars_list.index('Hyundai Elantra'). The output displays the value 4.* [Video description ends]

This we've verified by using the index function on the list and passing in Hyundai Elantra. Lists in Python also support a remove method. Simply call complete_cars_list.remove and specify the element that you want to remove.

[Video description begins] *The 16th code cell displays the following statement: complete_cars_list.remove('Hyundai Elantra'). It is followed by the second statement: complete_cars_list.* [Video description ends]

Remember, this element has to be an exact match with the element as it's present within the list. So here we've removed the Hyundai Elantra.

[Video description begins] *The output of the 16th code cell displays the car names in the following order: Toyota Camry, Honda Accord, Honda Civic, Toyota Corolla, Nissan Altima, Nissan Sentra, Kia Optima, Kia Soul, Hennessey Venom GT, Bugatti Veyron, and Koenigsegg Agera R. They are separated by commas and listed within the square brackets. The element Hyundai Elantra is missing from this list. The screen displays the* *original complete_cars_list statement towards center-right. The car entry Hyundai Elantra is highlighted in that list*. [Video description ends]

And if you look at the list of cars right now, you'll find that the Hyundai Elantra is no longer on this list. Now if you try to call the index function and pass in an element which has been removed, this is an error.

[Video description begins] *The 17th code cell displays the following statement: complete_cars_list.index('Hyundai Elantra'). The following error message displays below the code cell: ValueError: 'Hyundai Elantra' is not in list*. [Video description ends]

This error says, Hyundai Elantra is not in the list, which is correct.

Performing Useful List Operations

[Video description begins] *Topic title: Performing Useful List Operations. Your host for this session is Janani Ravi*. [Video description ends]

Before we go on to the next list operation, let's take a look at a slightly different way to perform an arithmetic operation using variables in Python.

[Video description begins] *A jupyter notebook titled MethodsInList-01 displays in the browser window. The page URL is localhost:8888/notebooks/MethodsInList-01%20ipynb. The main notebook area contains many empty code cells with the label In [ ] to their left*. [Video description ends]

Let's assume that a is an integer with a value of 10. If you want to add 1 to the existing value of a and store the result in a itself, there are two ways you can do this.

[Video description begins] *Following statements display at the bottom left corner of the screen: a += 1 and a= a + 1*. [Video description ends]

You can say a + = 1, your Python will interpret this as take the current value of a, add 1 to it, and store the result in a so that a will now be 11. This is the same as a = a+1. The syntax might seem peculiar, but this is supported by almost all programming languages, and this is something that you should be familiar with. It's not just the plus operation that can be used this way.

[Video description begins] *She refers to the a += 1 statement, which is highlighted*. [Video description ends]

Let's say, you wanted to give someone a 10X raise.

[Video description begins] *Following statements display at the bottom left corner of the screen: salary *= 10 and salary = salary * 10*. [Video description ends]

You can say, salary multiplication equal to 10. This will multiply the current salary that they have by 10, and assign the result to the salary variable. You can use the same operation in list as well.

[Video description begins] *The 18th code cell displays the following statement: complete_cars_list += ['Chevrolet Cruze','Chevrolet Malibu']. It is followed by the second statement: complete_cars_list*. [Video description ends]

We have the complete_cars_list += a new list, which contains the Chevrolet Cruze and the Chevrolet Malibu. This plus equal to operation will append this new list to the end of the complete_cars_list. And if you take a look at complete cars, you'll find the Chevrolet Cruze and Malibu at the very end.

[Video description begins] *The output of the 18th code cell displays the car names in the following order: Toyota Camry, Honda Accord, Honda Civic, Toyota Corolla, Nissan Altima, Nissan Sentra, Kia Optima, Kia Soul,* *Hennessey Venom GT, Bugatti Veyron, Koenigsegg Agera R, Chevrolet Cruze, and Chevrolet Malibu. They are separated by commas and listed within the square brackets. The screen displays the highlighted complete_cars_list statement towards center-right. It contains all the cars listed earlier except Chevrolet Cruze and Chevrolet Malibu.* [Video description ends]

Lists in Python have a whole host of useful methods that you can invoke.

[Video description begins] *The 19th code cell displays the following statement: complete_cars_list.sort( ). It is followed by the second statement: complete_cars_list.* [Video description ends]

If you want to sort all of the elements within your list, you simply call the sort operation or the sort function. And here is the complete_cars_list that has been sorted by default in the lexicographical order. If the list had contain numeric values, the default sort would be in the ascending order.

[Video description begins] *The output of the 19th code cell displays the car names in the following order: Bugatti Veyron, Chevrolet Cruze, Chevrolet Malibu, Hennessey Venom GT, Honda Accord, Honda Civic, Kia Optima, Kia Soul, Koenigsegg Agera R, Nissan Altima, Nissan Sentra, Toyota Camry, and Toyota Corolla. They are separated by commas and listed within the square brackets. The screen displays the highlighted complete_cars_list towards center-right. It contains all the car names in the order they were added.* [Video description ends]

There is also a handy little method on the list called reverse, which allows you to reverse the current order of elements in the list.

[Video description begins] *The 20th code cell displays the following statement: complete_cars_list.reverse( ). It is followed by the second statement: complete_cars_list.* [Video description ends]

Whatever the current order, reverse will simply change the list around so that the list are now in the reverse order as you can see here on screen.

[Video description begins] *The output of the 20th code cell displays the car names in the descending order. Toyota Corolla is the first element and Bugatti Veyron is the last. An upward arrow displays next to the list. The screen displays the highlighted complete_cars_list towards center-right. It contains all the car names in the ascending order. A downward arrow displays next to this list.* [Video description ends]

Toyota Corolla is now the first element in the list, it was previously the last. The last element of the list is the Bugatti Veyron, it was previously the first. Now on this reverse list, let's perform the pop operation.

[Video description begins] *The 21st code cell displays the following statement: complete_cars_list.pop( ). It is followed by the second statement: complete_cars_list.* [Video description ends]

The pop operation removes the last element in the list. If you look at the complete_cars_list, you'll find that the Bugatti Veyron, which was at the very last position has disappeared. It has been removed from the list, thanks to the pop function.

[Video description begins] *The output of the 21st code cell displays the car names in the descending order as earlier but the last element, Bugatti Veyron, is missing from the list. The screen displays the highlighted complete_cars_list towards center-right. It contains all the car names before the pop function, in the descending order.* [Video description ends]

The Chevrolet Cruze is currently the last element in this list. Now if you call pop on this list once again, the current last element will be removed.

[Video description begins] *The 22nd code cell displays the following statement: complete_cars_list.pop( ). It is followed by the second statement: complete_cars_list.* [Video description ends]

And if you take a look at the complete_cars_list, you will find that the Chevrolet Cruze has disappeared.

[Video description begins] *The last element in the output of the 22nd code cell is Chevrolet Malibu, Chevrolet Cruze is missing from the list.* [Video description ends]

The Chevrolet Malibu is now the new last element of this list. I've already mentioned earlier that list can contain duplicate elements.

[Video description begins] *The 23rd code cell displays the following statement: complete_cars_list.append ('Nissan Altima'). It is followed by the second statement: complete_cars_list. Both the statements are highlighted. There is an arrow pointing to the element: Nissan Altima.* [Video description ends]

The Nissan Altima is already present in our list. You can append another element, the Nissan Altima once again. And you can see in the resulting list that this car is present twice.

[Video description begins] *The output of the 23rd code cell displays the car names in the following order: Toyota Corolla, Toyota Camry, Nissan Sentra, Nissan Altima, Koenigsegg Agera R, Kia Soul, Kia Optima, Honda Civic, Honda Accord, Hennessey Venom GT, Chevrolet Malibu, and Nissan Altima. They are separated by commas and listed within the square brackets. The Nissan Altima elements are highlighted in the output.* [Video description ends]

Lists have this handy method called count, allowing you to count the number of times an element occurs in a list. Let's count the number of times the Honda Civic occurs in our list. Notice that the h is in lowercase, which is why we get the result 0.

[Video description begins] *The 24th code cell displays the following statement: complete_cars_list.count ('honda Civic'). The alphabet "h" is in the small case whereas in the list it was in uppercase. The output displays the value 0.* [Video description ends]

If we specify the h in uppercase and count the number of times the Honda Civic occurs, you'll see that the result is 1.

[Video description begins] *The 25th code cell displays the following statement: complete_cars_list.count ('Honda Civic'). The alphabet "H" is in uppercase. The output displays the value 1.* [Video description ends]

Now, let's do this with the Nissan Altima. Remember, the Nissan Altima is present as two different elements. And the count tells us that this element is present twice in the list.

[Video description begins] *The 26th code cell displays the following statement: complete_cars_list.count ('Nissan Altima'). The output displays the value 2. The screen displays the red highlighted complete_cars_list towards center-right and the Nissan Altima elements are highlighted in yellow.* [Video description ends]

The presence of duplicate elements is what makes a Python list different from a Python set. And you can convert this list to a set by simply passing in this list to the set function.

[Video description begins] *The 27th code cell displays the following statement: set(complete_cars_list).* [Video description ends]

This will convert the complete_cars_list to a set observed that the Nissan Altima is present exactly once, the duplicate entry has been removed in the set.

[Video description begins] *The output displays the car entries in the complete_cars_list as follows: Chevrolet Malibu, Hennessey Venom GT, Honda Accord, Honda Civic, Kia Optima, Kia Soul, Koenigsegg Agera R, Nissan Altima, Nissan Sentra, Toyota Camry, and Toyota Corolla. They are separated by commas and listed within curly brackets. The element Nissan Altima occurs once.* [Video description ends]

Also, notice that the Python set is shown with all of its elements enclosed in curly braces as opposed to square brackets. We'll study sets in more detail later on. Let's move on with list and understand some other functions that we can perform on list. Here is our cars_list, with all of its elements. If you want to remove all of the elements in one go from a list, you simply call the clear() function on the list, cars_list.clear will get rid of all of the elements.

[Video description begins] *The 28th code cell displays the following statement: print(cars_list). The output displays the car names in the following order: Toyota Camry, Honda Accord, Honda Civic, Toyota Corolla, Hyundai Elantra, Nissan Altima, Nissan Sentra, Kia Optima, and Kia Soul. They are separated by commas and listed within the square brackets.* [Video description ends]

And if you print out cars_list, you'll see that it's now an empty list, denoted just by the square brackets.

[Video description begins] *The 29th code cell displays the following statement: cars_list.clear( ). The 30th code cell displays cars_list and its output displays: [ ].* [Video description ends]

Clearing the cars_list, of course, does not affect the complete_cars_list, which was created using a concatenation of cars_list and cars_list2.

[Video description begins] *The 31st code cell displays the following statement: complete_cars_list. The output displays the car names in the following order: Toyota Corolla, Toyota Camry, Nissan Sentra, Nissan Altima, Koenigsegg Agera R, Kia Soul, Kia Optima, Honda Civic, Honda Accord, Hennessey Venom GT, Chevrolet Malibu, and Nissan Altima. They are separated by commas and listed within the square brackets.* [Video description ends]

The complete_cars_list still has all of its elements. The Nissan Altima is present twice. If you have an original list and you want to create a copy of this list, you can call the copy function on the list and assign the new list to a variable. Here, the variable is the new_cars_list.

[Video description begins] *The 32nd code cell displays the following statement: new_cars_list = complete_cars_list.copy( ).* [Video description ends]

Now you should know that the copy function performs a deep copy of the list. A deep copy essentially means that this new_cars_list is an entirely new list. The list itself and all of the elements are copied over to a new portion in memory. When you create a deep copy of a list, any changes that you make to the original list does not affect the copy.

The two lists are now entirely separate. Let's see how the lists are different here. Once I've created this new_cars_list, which is a deep copy of the complete_cars_list, I'm going to go ahead and call the del keyword with the complete_cars_list.

[Video description begins] *The 33rd code cell displays the following statement: del complete_cars_list.* [Video description ends]

The del keyword deletes the complete_cars_list, and removes all of its contents from memory. When you execute this del command on the complete_cars_list, the original list will completely disappear from memory. If you try to access the complete_cars_list after you've deleted it from memory using the del command, you'll see that this is an error, the complete_cars_list is no longer defined. Python no longer knows of its existence.

[Video description begins] *The 34th code cell displays the following statement: complete_cars_list. The following error message displays below the code cell: NameError: name 'complete_cars_list' is not defined.* [Video description ends]

Deleting the complete_cars_list, however, has no effect on the deep copy that we had made previously. If you take a look at the new_cars_list, you'll find that all of the list, and its contents are still available for us to use. The deep copy is a completely different copy changes to the original to not affect your copy. It's also possible in Python to create a shallow copy of a list.

[Video description begins] *The 35th code cell displays the following statement: new_cars_list. The output displays the same car names that were previously in the complete_cars_list.* [Video description ends]

And the shallow copy is created by simply using the assignment operator.

[Video description begins] *The 36th code cell displays the following statement: another_cars_list = new_cars_list. It is followed by the second statement: another_cars_list.* [Video description ends]

another_cars_list is a shallow copy of the new_cars_list. Here, under the hood in Python, the two variables another_cars_list, and new_cars_list both refer to the same list. This is a shallow copy. When you create a shallow copy of a list, the changes that you make using one of the variables, which holds the shallow copy will be reflected in the other list as well. Under the hood, they both refer to the same list.

[Video description begins] *The output displays the same list that was displayed for new_cars_list. The screen also displays the new_cars_list to the right and both the lists are highlighted.* [Video description ends]

Here are the contents of another_cars_list, which is a shallow copy of the contents in the new_cars_list. I'm now going to invoke the remove function on another_cars_list. I'm going to remove the Hennessey Venom GT.

[Video description begins] *The 37th code cell displays the following statement: another_cars_list.remove('Hennessey Venom GT'). The 38th code cell displays the following statement: another_cars_list.* [Video description ends]

And if you take a look at another_cars_list, you will find that this car has disappeared.

[Video description begins] *The output of 38th code cell displays the elements of another_cars_list minus the Hennessey Venom GT element.* [Video description ends]

Now if you remember, another_cars_list is a shallow copy of the new_cars_list. And if you look at the contents of the new_cars_list, the Hennessey Venom GT is not present here as well. Changes that we made to another_cars_list are reflected in the new_cars_list as well because under the hood, they are the same list.

[Video description begins] *The 39th code cell displays the following statement: new_cars_list. The output displays the same car names that were previously displayed for another_cars_list and the Hennessey Venom GT element is missing.* [Video description ends]

I'll now use the new_cars_list variable and remove the Nissan Sentra from the list.

[Video description begins] *The 41st code cell displays the following statement: new_cars_list.remove('Nissan Sentra'). It is followed by the second statement: new_cars_list.* [Video description ends]

If you take a look at the new_cars_list, the Nissan Sentra car is no longer present. If you take a look at another_cars_list, which if you remember is a shallow copy of the new_cars_list, the Nissan Sentra has disappeared from this list as well.

[Video description begins] *The output displays the contents of the new_cars_list. The Nissan Sentra element has been removed.* [Video description ends]

So if you're working with list copies, make sure you specify the right kind of copy for your use case.

[Video description begins] *The 42nd code cell displays the following statement: another_cars_list. The output displays the same car names that were previously displayed for new_cars_list and the Nissan Sentra element is also missing from this list.* [Video description ends]

A deep copy or a shallow copy.

Using Built-in Functions with Lists

[Video description begins] *Topic title: Using Built-in Functions with Lists. Your host for this session is Janani Ravi*. [Video description ends]

Let's now see some other functions and operations that we can perform using list.

[Video description begins] *A jupyter notebook titled MethodsInList-02 displays in the browser window. The page URL is localhost:8888/notebooks/MethodsInList-02%20ipynb. The main notebook area contains many empty code cells with the label In [ ] to their left*. [Video description ends]

This time, we'll work with a list of numeric values.

[Video description begins] *The first code cell displays the following statement: list_num = [20 ,70, 80 ,50,30,60 ,40]. It is followed by the second statement: list_num*. [Video description ends]

In this case, these numeric values happen to be integers, but all of these operations will work with list of floats as well. We initialize the list_num variable, which is a list of integers enclosed in square brackets. Now let's call the max built-in function, which we've seen before.

[Video description begins] *The following output label and output display below it: Out [1]: [20 ,70, 80 ,50,30,60 ,40]*. [Video description ends]

But this time instead of passing the individual numbers to this function, let's pass in a list. And this max function works on a list as well, and will return the maximum element in our list, which in our case happens to be 80.

[Video description begins] *The second code cell displays the following statement: max(list_num). The output below displays the value 80*. [Video description ends]

Just like the max function, you can also invoke the min function and pass in a list as its input argument. And it will find the minimum number in our list, which here is 20.

[Video description begins] *The third code cell displays the following statement: min(list_num). The output below displays the value 20*. [Video description ends]

We've already seen earlier that the len function will give us the number of elements in our list. Our integer list contains a total of 7 elements. There are multiple ways that you can sort the elements of a list.

[Video description begins] *The fourth code cell displays the following statement: len(list_num). The output below displays the value 7. She highlights the list_num elements in the first Input code cell. The fifth code cell displays the following statement: list_num.sort( ). It is followed by the second statement: list_num*. [Video description ends]

We've seen how we can invoke the sort function on the list itself. This sort function will affect the list itself so that list_num now has all of its elements sorted in the ascending order. The list itself has changed.

[Video description begins] *The output of the fifth code cell displays the numbers in the following order: 20, 30, 40, 50, 60, 70, and 80. They are separated by commas and listed within the square brackets. She highlights the list_num elements in the first Input code cell to show the original sequence of numbers*. [Video description ends]

Now let's go ahead and append a new element to the very end of this list. I'm going to add 25 to this list. So list_num is no longer in the sorted order.

[Video description begins] *The sixth code cell displays the following statement: list_num.append(25). It is followed by the second statement: list_num. The output of the sixth code cell displays the numbers in the following order: 20, 30, 40, 50, 60, 70, 80, and 25. They are separated by commas and listed within the square brackets. The screen also displays the earlier sorted list_num elements on the right-hand side.* [Video description ends]

You can use the built-in sorted function to sort the elements of a list.

[Video description begins] *The seventh code cell displays the following statement: sorted_list_num = sorted(list_num).* [Video description ends]

And the difference when you use the sorted function is the fact that the original list is not affected. The sorted function creates a brand new list with the elements in sorted order, ascending order by default.

[Video description begins] *The eighth code cell displays the following statement: sorted_list_num. The following output displays below: 20, 25, 30, 40, 50, 60, 70, and 80. They are separated by commas and listed within the square brackets.* [Video description ends]

The sorted built-in function returns a new sorted list, which we stored in the sorted _list_num variable. If you take a look at the original list_num variable, you'll see that the element 25 is at the very end. The original list hasn't been affected by the sort.

[Video description begins] *The ninth code cell displays the following statement: list_num. The output displays the numbers in the following order: 20, 30, 40, 50, 60, 70, 80, and 25. They are separated by commas and listed within the square brackets. She highlights the list_num elements in the output of the sixth code cell to show that both outputs are same and list_num has not changed.* [Video description ends]

There are other useful built-in functions that operate on lists as well, such as the sum function.

[Video description begins] *The tenth code cell displays the following statement: sum(list_num).* [Video description ends]

When you pass in a list of numbers, the sum function will simply sum up all of the elements of the list and give you a result which in our case is 375.

[Video description begins] *The output displays the value 375, which is highlighted in yellow. The output of code cell 9 is highlighted in red.* [Video description ends]

There are also other interesting arithmetic kind of operations you can perform on list. Here is a combination of the multiplication and the assignment operator. So we want to take list_num, multiply it by 2, and assign the result to list_num itself.

[Video description begins] *The 11th code cell displays the following statement: list_num *= 2. Following text displays at the bottom left corner of the screen: list_num *= 2 is the same as list_num = list_num * 2.* [Video description ends]

But what does it mean to multiply a list by 2? If you take a look at the result, you'll see that it means you simply take all of the original elements of list_num, and concatenated again, and show the result in list_num.

[Video description begins] *The 12th code cell displays the following statement: list_num. The output displays the numbers in the following order: 20, 30, 40, 50, 60, 70, 80, 25, 20, 30, 40, 50, 60, 70, 80, and 25. They are separated by commas and listed within the square brackets. She highlights and labels the first set and the repeated set of numbers as 1 and 2 respectively. The previous list_num displays to the center-right of the screen for comparison purpose.* [Video description ends]

So list_num now contains all of the elements twice. You'll see that this kind of multiplication or arithmetic operation on a list only works with integers. If you try to multiply by a floating point value, this is meaningless, and Python will throw an error.

[Video description begins] *The 13th code cell displays the following statement: list_num *= 2.0. The following error message displays below the code cell: TypeError: can't multiply sequence by non-int of type 'float'*. [Video description ends]

Before we move on to the next example, you should know that in Python a numeric value of zero is considered to be the equivalent of the Boolean False, and all numeric values greater than zero are considered to be True. The all function, when it receives a list as its input argument, will check to see whether all elements in a list are True.

[Video description begins] *The 14th code cell displays the following statement: all([0])*. [Video description ends]

Here we have a list with just a single element that is 0. The all built-in function will check the single element, find that this is zero. Remember, zero is the equivalent of False. It says all the elements of this list are False, and that is why the return value here is False.

[Video description begins] *She refers to the output of the 14th code cell that displays the text: False*. [Video description ends]

Let's say we pass in a list with a single element with a value 1 to the all function. Remember, any non-zero value in Python is considered True. So the all function when it receives this list with a single element, which is non-zero will say, yes, all of the elements of this list are True.

[Video description begins] *The 15th code cell displays the following statement: all([1]). The output displays the text: True*. [Video description ends]

We'll now pass in a list with two elements, one of these elements is 0. The other element is 1, we'll pass this list into the all function, and this will rightfully return False. Because all of the elements are not True, there is zero in there.

[Video description begins] *The 16th code cell displays the following statement: all([0, 1]). The output displays the text: False*. [Video description ends]

In the case of the list_num list if you remember, all of the elements are greater than zero.

[Video description begins] *The 17th code cell displays the following statement: all(list_num). The list_num list displays to the center-right of the screen*. [Video description ends]

When you pass in this list to the all function, this function will return True, indicating that all of the elements of this list are non-zero elements. If you have an empty list with no elements and you invoke the all function on this list, the return value will be True.

[Video description begins] *The 18th code cell displays the following statement: new_num_list = [ ]. It is followed by the second statement: all(new_num_list). The output displays the text: True*. [Video description ends]

Just like the all function, there is another built-in function in Python, the any function. The any function will check to see whether any element in this list is True.

[Video description begins] *The 19th code cell displays the following statement: any([0])*. [Video description ends]

When you pass in a single element list which only has the number 0, any will return False.

[Video description begins] *She refers to the output of the 19th code cell that displays the text: False*. [Video description ends]

Because there is no element in this list that is True, the only element is 0, which is considered False. If you pass in a list with just one element, which is a non-zero element to the any function, the return value will be True as you see here on screen.

[Video description begins] *She refers to the 20th code cell that displays the following statement: any([1]). The output displays the text: True*. [Video description ends]

Let's try the any function on this list with two elements, one of which is 0 and the other that is non-zero.

[Video description begins] *The 21st code cell displays the following statement: any([0, 1])*. [Video description ends]

At least one element is non-zero, which is why any returns True. Let's invoke the any function on our list number, all of the numbers are greater than zero which is why any returns True.

[Video description begins] *The 22nd code cell displays the following statement: any(list_num). The list_num list values display to the center-right of the screen for reference purpose. The output displays the text: True*. [Video description ends]

Now let's invoke the any function on an empty list that is the new_num_list, and any returns False.

[Video description begins] *The 23rd code cell displays the following statement: any(new_num_list). The contents of the 18th Input code cell and its output are displayed on the right side of the screen for reference purpose*. [Video description ends]

This might seem a little strange to you. But you should understand how all of these built-in functions work with empty lists and with other edge cases so that you're not thrown when you're writing real code.

Perform Slicing Operations on Lists

[Video description begins] *Topic title: Perform Slicing Operations on Lists. Your host for this session is Janani Ravi. A jupyter notebook titled ListSlicingAndListOperations displays in the browser window. The page URL is localhost:8888/notebooks/ListSlicingAndListOperations%20ipynb. The main notebook area contains many empty code cells with the label In [ ] to their left*. [Video description ends]

In this demo we will continue working with list operations with a specific focus on list slicing.

[Video description begins] *The first Input code cell displays the following statement: my_car_list = ['Toyota Corolla', 'Toyota Camry', 'Nissan Altima', 'Kia Soul', 'Kia Optima', 'Honda Civic']. It is followed by the second statement: my_car_list*. [Video description ends]

We'll talk about what exactly it is in just a bit. Let's initialize a new list of cars. I call it my_car_list, and it contains a total of six cars.

[Video description begins] *The following output label and output display below the code cell: Out [1]: ['Toyota Corolla', 'Toyota Camry', 'Nissan Altima', 'Kia Soul', 'Kia Optima', 'Honda Civic']*. [Video description ends]

Let's use the len function in order to confirm the length of this list of cars. And you can see that len returns 6. We are all set here.

[Video description begins] *The second code cell displays the following statement: len(my_car_list). The output below displays the value 6*. [Video description ends]

Let's move on and access individual cars from this list by using their index position values. my_car_list[4], where 4 is specified in square brackets, will access the fifth car in the list, which is the Kia Optima.

[Video description begins] *The third code cell displays the following statement: my_car_list [4].* [Video description ends]

Remember that the first element in a Python list is at index zero.

[Video description begins] *The output for third code cell displays: 'Kia Optima'.* [Video description ends]

Let's try this once again. And let's access the element at index 5. And you see that this is the Honda Civic. This is the sixth element in the list. We've seen that Python list work with negative indices as well.

[Video description begins] *The fourth code cell displays the following statement: my_car_list [5]. The output displays: 'Honda Civic'.* [Video description ends]

We simply count the elements starting with the very end of the list.

[Video description begins] *The fifth code cell displays the following statement: my_car_list [-2]. The output displays: 'Kia Optima'.* [Video description ends]

So my_car_list[-2] will give us Kia Optima, which is the second to last element in the list.

[Video description begins] *The my_car_list elements display on the right side of the screen for reference purpose and Kia Optima is highlighted.* [Video description ends]

In exactly the same way, my_car_list of index -4 will give us the fourth to last element, which is the Nissan Altima.

[Video description begins] *The sixth code cell displays the following statement: my_car_list [-4]. The output displays: 'Nissan Altima'.* [Video description ends]

Let's take a look at the contents of a car_list before we move on to more complex list slicing operations.

[Video description begins] *The seventh code cell displays the following statement: my_car_list. The output displays the car names in the following order: Toyota Corolla, Toyota Camry, Nissan Altima, Kia Soul, Kia Optima, and Honda Civic. They are separated by commas and listed within the square brackets.* [Video description ends]

Instead of accessing single elements in a list, list slicing operations allow us to access a range of elements. List slicing operations in Python can get pretty complex. They're hard to read and understand, but they're used very often, which is why I'll walk you through these step-by-step. Let's perform our first list slicing operation.

[Video description begins] *The eighth code cell displays the following statement: my_car_list [0 : 4]. Following statement is displayed at the bottom-left corner of the screen: list_name [start_index : end_index].* [Video description ends]

Here I've specified two numbers within square brackets separated by a colon. And this number represents a range of elements in the list, start _index : end_ index. We want to access all of the list elements starting at index zero going up to index four.

[Video description begins] *There are two blue-colored unidirectional arrows pointing from start_index to the value 0 and from end_index to the value 4 in the 8th code cell.* [Video description ends]

The end index is exclusive, we want to include the element at the start index. This slicing operation will give us the elements at index positions zero, one, two and three, which are essentially the first four elements in our list.

[Video description begins] *The output displays the car names in the following order: Toyota Corolla, Toyota Camry, Nissan Altima, and Kia Soul. They are separated by commas and listed within the square brackets*. [Video description ends]

Let's try another example of a slicing operation.

[Video description begins] *The ninth code cell displays the following statement: my_car_list [0 : 2]*. [Video description ends]

Here I want the elements of the list at index positions, zero, and one. So I specify the range 0:2. This returns the first two elements of the list, Toyota Corolla and the Toyota Camry. If you want to get all of the elements in the list, simply specify the start index as 0 and the end index as the length of the list. And this will return all of the original elements in the list.

[Video description begins] *The tenth code cell displays the following statement: my_car_list [0 : 6]. The my_car_list elements display on the right side of the screen for reference purpose. The output displays all the elements in the my_car_list*. [Video description ends]

Now you should remember that list slicing operations creates a deep copy of the list. If you store the result of your slicing operation in a variable, what you have is a deep copy. It's an entirely new list, and changes to the original list will not affect the deep copy if you remember. You can use list slicing to get all of the original elements in a list by specifying an end index that is beyond the number of elements that you have in a list.

[Video description begins] *The screen also displays the statement: Creates a new list with the range specified*. [Video description ends]

We have a list with just six elements. The end index that we have specified here is 7. Since our start index is 0 an entire copy of the list is returned. The start index in a list slicing operation can be any value other than zero.

[Video description begins] *The 11th code cell displays the following statement: my_car_list [0 : 7]. The output displays all the elements in the my_car_list*. [Video description ends]

And we start at that index position. If you specify an index position beyond the number of elements you have in a list, what you get is an empty list as you see here on screen.

[Video description begins] *The 12th code cell displays the following statement: my_car_list [7 :]. The output displays [ ]*. [Video description ends]

You can specify any sub range within your list as a part of your slicing operation. Here, we will return the elements starting at index position 5 and going through to 6. The element at index position 6 will not be included, which means we only return the last element in the list. This is the element at index 5.

[Video description begins] *The 13th code cell displays the following statement: my_car_list [5 : 6]. The output displays ['Honda Civic']. The my_car_list elements display on the right side of the screen for reference purpose*. [Video description ends]

Let's see some more examples of list slicing.

[Video description begins] *The 14th code cell displays the following statement: my_car_list [3 : 6]*. [Video description ends]

Here, we want to access all of the elements, starting at index 3, and up to but not including index 6.

[Video description begins] *The following elements are highlighted in the reference list: Kia Soul, Kia Optima, and Honda Civic. Two blue-colored arrows point to values 3 and 6 in the code cell. At the same time, two blue-colored arrows point to the Kia Soul and Honda Civic elements in the reference list*. [Video description ends]

This gives us the Kia Soul, Kia Optima, and Honda Civic from our original list.

[Video description begins] *She refers to the output of code cell 14.* [Video description ends]

It's not necessary within a slicing operation to specify both the start index as well as the end index. You can omit one of these values, or even both of these values, and Python will infer what these values should be.

[Video description begins] *The 15th code cell displays the following statement: my_car_list [: 3].* [Video description ends]

When you omit just the start index it'll start at index 0. So specifying just colon and 3 will give us the cars at indices zero, one and two.

[Video description begins] *The output displays the first three car names in the following order: Toyota Corolla, Toyota Camry, and Nissan Altima. They are separated by commas and listed within the square brackets. These first three elements are also highlighted in the reference list displayed on the right side of the screen.* [Video description ends]

If you omit the end index and only specify the start index, Python will infer that you want to go through till the very end of the list.

[Video description begins] *The 16th code cell displays the following statement: my_car_list [3 :].* [Video description ends]

So here we'll start with the element at index 3 and list all of the elements up to the very end of the list.

[Video description begins] *The output displays the last three car names in the following order: Kia Soul, Kia Optima, and Honda Civic. They are separated by commas and listed within the square brackets. These last three elements are also highlighted in the reference list displayed on the right side of the screen.* [Video description ends]

Honda Civic is the last element. List slicing works with negative indices as well. And the way Python infers these negative indices can be a little confusing.

[Video description begins] *The 17th code cell displays the following statement: my_car_list [0 : -4].* [Video description ends]

Remember that the index position -4 refers to the fourth to last element in the list. So if you go from 0 to -4, you're starting from the first index in the list and going up to the fourth last element in the list. Which is why this result is the Toyota Corolla and the Toyota Camry.

[Video description begins] *The output displays the first two car names in the following order: Toyota Corolla and Toyota Camry. They are separated by commas and listed within the square brackets. The first two elements are highlighted in the reference list displayed on the right side of the screen. There is also a blue-colored arrow pointing to the third element, Nissan Altima.* [Video description ends]

The fourth to last element in our list is the Nissan Altima car, which comes right after the Toyota Camry. Remember the end index is excluded. Both of your start as well as n index values can be negative, so -4 colon -1 is completely valid.

[Video description begins] *The 18th code cell displays the following statement: my_car_list [-4 : -1].* [Video description ends]

This statement means that you want to access the list starting from the fourth to last element in the list. Which is the Nissan Altima, and up to but not including the last element in the list indicated by -1.

[Video description begins] *There are two blue-colored unidirectional arrows pointing to the values -4 and -1 in the 18th code cell. The elements Nissan Altima and Honda Civic are highlighted in the list for reference purpose*. [Video description ends]

Which is why you get the Nissan Altima, the Kia Soul and the Kia Optima.

[Video description begins] *The output displays the following elements: Nissan Altima, Kia Soul, and Kia Optima. They are separated by commas and listed within the square brackets. These elements are also highlighted in the reference list displayed on the right side of the screen. There is a blue-colored arrow pointing to the last element, Honda Civic*. [Video description ends]

The Honda Civic, which is the last element in the list, is not included. If the range you specify in the list slicing operation does not make sense, Python will return an empty list.

[Video description begins] *The 19th code cell displays the following statement: my_car_list [-2 : -4]*. [Video description ends]

Here, the start index is -2, so we start from the second to last position in the list and go till the fourth to last position in the list.

[Video description begins] *There is a blue-colored unidirectional arrow pointing to the value -2 in the 19th code cell. The element Kia Optima is highlighted in the list for the reference purpose to indicate it is the element at position -2*. [Video description ends]

This is not a range that makes sense to Python because the start index is after the end index, which is why Python returns an empty list.

[Video description begins] *The output displays [ ]*. [Video description ends]

Using Step Size in Slicing Operations

[Video description begins] *Topic title: Using Step Size in Slicing Operations. Your host for this session is Janani Ravi*. [Video description ends]

We'll continue working with the my_car_list, which has six car names within it.

[Video description begins] *A jupyter notebook titled ListSlicingAndListOperations displays in the browser window. The Input code cell 18 displays the following statement: my_car_list [-4 : -1]. The 19th code cell displays the following statement: my_car_list [-2 : -4]. The 20th code cell displays the following statement: my_car_list. The following output displays for code cell 20: ['Toyota Corolla', 'Toyota Camry', 'Nissan Altima', 'Kia Soul', 'Kia Optima', 'Honda Civic']*. [Video description ends]

Let's see some more advanced slicing operations here. This time we'll specify not just the start index and the end index, but we'll use a colon and specify a step size as well.

[Video description begins] *The 21st code cell displays the following statement: my_car_list [0:5:]. Following statement is displayed at the bottom-left corner of the screen: list_name [start_index:end_index:step_size]. Following text displays below: step_size is the number of elements to step over when accessing the list*. [Video description ends]

Here, we omitted the step size. The step size is assumed to be 1 by default.

[Video description begins] *She points to space after 5:*. [Video description ends]

The step size in our slicing operation tells Python how we want to access the range that we have specified using the start and end indices. Here the start index is 0, the end index is 5. And the step size by default, is considered to be 1.

[Video description begins] *The output displays the elements in the following order: Toyota Corolla, Toyota Camry, Nissan Altima, Kia Soul, and Kia Optima. They are separated by commas and listed within the square brackets. The my_car_list elements display on the right side of the screen for reference purpose. First five elements are highlighted in the reference list.* [Video description ends]

Which means we want to access all of the elements starting at index position 0 and up to index position 5, but not including 5. This slicing operation gives us the elements at indices 0, 1, 2, 3, and 4. Let's try another slicing operation. Here, we omitted both the start as well as the end index values and have specified a step size of 2.

[Video description begins] *The 22nd code cell displays the following statement: my_car_list [: : 2].* [Video description ends]

This will allow us to access every alternate element in the list, starting from the very beginning and going up to the very end. Observe that this slicing operation gives us the element at index 0.

[Video description begins] *The output displays the elements in the following order: Toyota Corolla, Nissan Altima, and Kia Optima. They are separated by commas and listed within the square brackets. The my_car_list elements display on the right side of the screen for reference purpose. The alternate elements are highlighted and there is a small blue-colored arrow pointing to Toyota Camry and Kia Optima.* [Video description ends]

That is the Toyota Corolla. Then we skip over one element, and get the Nissan Altima. Skip over the Kia Soul, get the Kia Optima. And that is the end. Let's use the same step size 2. But this time we specify a range. We want to start at index 0 and go up to but not include index 5.

[Video description begins] *The 23rd code cell displays the following statement: my_car_list [0: 5:2].* [Video description ends]

Notice that the result has the Toyota Corolla at index position 0. That's where we start.

[Video description begins] *The output displays the elements in the following order: Toyota Corolla, Nissan Altima, and Kia Optima. They are separated by commas and listed within the square brackets.* [Video description ends]

We skip over the Camry, have the Nissan Altima. Skip over the Soul, and have the Optima. At index position 5, we have the Honda Civic. That is not included.

[Video description begins] *There is a blue-colored arrow pointing to the value 5 in the 23rd code cell and another blue-colored arrow pointing to Honda Civic in the reference list.* [Video description ends]

We'll now use the same step size 2, but we'll start at index 2. That is our start index and we go up to, but not include 5. The result here is the Nissan Altima and the Kia Optima.

[Video description begins] *The 24th code cell displays the following statement: my_car_list [2:5:2]. The output displays the elements in the following order: Nissan Altima and Kia Optima. They are separated by commas and listed within the square brackets.* [Video description ends]

The Nissan Altima is the car at index position 2. We skip over the Kia Soul, and have the Kia Optima included as well.

[Video description begins] *She refers to the elements in the reference list where Nissan Altima and Kia Optima are highlighted.* [Video description ends]

Depending on what elements of a list you want to access, you can specify a different value for the step size. For example, here we have a step size of 3.

[Video description begins] *The 25th code cell displays the following statement: my_car_list [0: 5:3]. The output displays the elements in the following order: Toyota Corolla and Kia Soul. They are separated by commas and listed within the square brackets.* [Video description ends]

We start at index position 0, so we have the Toyota Corolla. We skip over two other elements, Camry and the Altima. And we have the Kia Soul in the result. Let's try one more example of list slicing using the step size. We start at index 1, go up to but do not include index 4. The step size is 2.

[Video description begins] *The 26th code cell displays the following statement: my_car_list [1: 4:2].* [Video description ends]

What do you think the result will be? Well, if you thought Toyota Camry and Kia Soul you would have been absolutely right.

[Video description begins] *The output displays the elements in the following order: Toyota Camry and Kia Soul. These two elements are also highlighted in the reference list.* [Video description ends]

Here is the car list that we've been working with, containing a total of six cars.

[Video description begins] *The 27th code cell displays the my_car_list statement followed by the output that displays all the six car names.* [Video description ends]

Now the step size can also have negative values. Specifying negative values allows you to access the list in some pretty interesting ways. Here, the start index and the end index have both been omitted.

[Video description begins] *The 28th code cell displays the following statement: my_car_list [: : -1].* [Video description ends]

So we go from the very beginning of our list to the very end. But we step through this using -1.

[Video description begins] *The output displays the elements in the following order: Honda Civic, Kia Optima, Kia Soul, Nissan Altima, Toyota Camry, and Toyota Corolla. They are separated by commas and listed within the square brackets. The my_car_list elements displayed on the right side of the screen for reference purpose have an upward arrow to their left.* [Video description ends]

This will basically access our list in the reverse order. And this is a simple command you can use to reverse the elements in a list. So you can use the start and end index values to specify what portion of the list you're interested in, and -1 will step through the list in the reverse order. Here, we start with index 2 go down to 0. But we want to access it in the reverse order. Which means we get the Nissan Altima first and then the Toyota Camry.

[Video description begins] *The 29th code cell displays the following statement: my_car_list [2: 0 : -1]. The output displays two elements: Nissan Altima and Toyota Camry. These two elements are also highlighted in the reference list and have a blue-colored upward arrow to their left.* [Video description ends]

Couple of more examples with negative step sizes should help cement your understanding.

[Video description begins] *The 30th code cell displays the following statement: my_car_list [4: 1 : -1].* [Video description ends]

Here, the start stock index is 4 end index is 1. But we want to step through the list backwards. Which is why we get the Kia Optima at index 4.

[Video description begins] *The output displays three elements: Kia Optima, Kia Soul, and Nissan Altima. These three elements are also highlighted in the reference list and have a blue-colored upward arrow to their left*. [Video description ends]

We move backwards through this list. We then get the Kia Soul, then the Nissan Altima. But we do not include the Toyota Camry, which is at index position 1. Remember, the end index is exclusive.

[Video description begins] *She points to the element Toyota Camry in the reference list*. [Video description ends]

Let's try out some more complex examples, so you get the hang of list slicing. Here the start index is -4.

[Video description begins] *The 31st code cell displays the following statement: my_car_list [-4: 0 : -1]*. [Video description ends]

That is the fourth to last element of this list, which is the Nissan Altima.

[Video description begins] *She points to the value -4 in the code cell and the element Nissan Altima is highlighted in the reference list*. [Video description ends]

We go up to index 0, but do not include index 0. And we step backwards, that is -1.

[Video description begins] *The output displays two elements: Nissan Altima and Toyota Camry. These two elements are also highlighted in the reference list and have a blue-colored upward arrow to their left*. [Video description ends]

So we get the Nissan Altima and the Toyota Camry in the result. Before we complete our discussion on list slicing operations, let's see one example where the step size is something other than -1.

[Video description begins] *The 32nd code cell displays the following statement: my_car_list [: : -2]*. [Video description ends]

Let's try -2. And we're interested in the entire list. We've omitted the start as well as the end index. This will access the elements of the list starting from the last element, that is the Honda Civic. The -1, we'll walk down the list backwards. We'll skip over the Kia Optima, access the Kia Soul.

[Video description begins] *The output displays three elements: Honda Civic, Kia Soul, and Toyota Camry. These three elements are also highlighted in the reference list and have a blue-colored upward arrow to their left. The elements Kia Optima and Nissan Altima have an arrow-shaped bullet to their right side to indicate that they have been skipped*. [Video description ends]

Skip over the Nissan Altima, access the Toyota Camry. And that is exactly what this list slice returns. Now let's move on. This is the car list that we are working with, with six elements.

[Video description begins] *The 33rd code cell displays the my_car_list statement followed by the output that displays all the six car names*. [Video description ends]

We can call print on this car list and len on this car list, to print out all six elements. The length of the list is 6. Now let's see one slightly funky operation.

[Video description begins] *The 34th code cell displays the following statement: print(my_car_list). It is followed by the statement: len(my_car_list). The output displays the car names, which are separated by commas and listed within the square brackets. Below them is the value 6*. [Video description ends]

Here, I'm going to assign two new elements to the car list.

[Video description begins] *The 35th code cell displays the following statement: my_car_list [6: -1] = ['Ford Focus', 'Dodge Charger']. It is followed by the statement: my_car_list.* [Video description ends]

And on the left-hand side, where I assign to the car list as specified, something that looks like a list slicing operation. The first element, the Ford Focus will be assigned to index 6.

[Video description begins] *Two blue-colored arrows point to the value 6 and the element Ford Focus respectively.* [Video description ends]

[Video description begins] *Two blue-colored arrows point to the value -1 and the element Dodge Charger respectively.* [Video description ends]

And the second element, the Dodge Charger will be assigned to index -1, which is basically the last element in the list. If you take a look at the car list now, you can see that the Ford Focus is at position 6. And the Dodge Charger is the last element in the list.

[Video description begins] *The output displays the elements in the following order: Toyota Corolla, Toyota Camry, Nissan Altima, Kia Soul, Kia Optima, Honda Civic, Ford Focus, and Dodge Charger. They are separated by commas and listed within the square brackets.* [Video description ends]

This is a rather funky way of assigning to a list. This is not often used, but you should know that such assignments are possible. I'll now quickly introduce the in command that you can use in Python. And this can be used to check whether a particular element is in a list.

[Video description begins] *The 37th code cell displays the following statement: 'Ford Focus' in my_car_list. All the elements of the my_car_list are displayed on the right side of the screen for reference purpose.* [Video description ends]

Is Ford Focus in my_car_list?

[Video description begins] *The output displays True. The Ford Focus element is highlighted on the right side.* [Video description ends]

Well, the return value is True because the Ford Focus is indeed present. Let's get rid of the Ford Focus element by invoking the pop function on the list.

[Video description begins] *The 38th code cell displays the following statement: my_car_list.pop(-2).* [Video description ends]

Pop, when we pass in an index value will pop the element at index position -2. So counting from the end of the list, this will pop the second to last element in the list, which is the Ford Focus.

[Video description begins] *The output displays the following: 'Ford Focus'.* [Video description ends]

Now if you try and use the in operation to check whether the Ford Focus is in my_car_list you'll find that the return value is False.

[Video description begins] *The 39th code cell displays the following statement: 'Ford Focus' in my_car_list. The output displays False.* [Video description ends]

Because this element is no longer present in my_car_list, as you can see by displaying the contents of my_car_list.

[Video description begins] *The 40th code cell displays the following statement: my_car_list. The output displays the elements in the following order: Toyota Corolla, Toyota Camry, Nissan Altima, Kia Soul, Kia Optima, Honda*

Civic, and Dodge Charger. They are separated by commas and listed within the square brackets. The element Ford Focus has been removed from the list. [Video description ends]

The Ford Focus has disappeared thanks to our pop statement.

Working with Strings as a List of Characters

[Video description begins] *Topic title: Working with Strings as a List of Characters. Your host for this session is Janani Ravi.* [Video description ends]

Now that we've understood how Python list function, we'll see that, a string in Python is essentially very similar to a list it can be thought of as a list of characters. Let's take a look at this string, W-O-R-L-D.

[Video description begins] *A jupyter notebook titled StringsAsListsAndStringFunctions displays in the browser window. The Input code cell 1 displays the following statement: print('World').* [Video description ends]

World here that we print out to screen.

[Video description begins] *The output displays the word: World.* [Video description ends]

Notice that World is made up of a sequence of characters, We assign the sequence of characters to the variable x, x contains a string.

[Video description begins] *The second code cell displays the following statement: x = 'World'. Below it is the statement: x. The output displays the word, World, within single quotes. The first alphabet, W, of the word World, is in uppercase and the remaining alphabets are in lowercase.* [Video description ends]

Just like the very first element of a list is at index position 0. You can access the very first character in a string by accessing index position 0 specified in square brackets.

[Video description begins] *The third code cell displays the following statement: x [0].* [Video description ends]

x[0] gives us W which is the first character in our sequence of characters.

[Video description begins] *The output displays the uppercase alphabet: W within single quotes.* [Video description ends]

x[3] will give is a character at index position 3, which is the fourth character l.

[Video description begins] *The fourth code cell displays the following statement: x [3]. The output displays the lowercase alphabet: l within single quotes.* [Video description ends]

The string World has exactly five characters. It has a length of 5, so if you try to access a character at index position 6, you'll get an error saying string index out of range.

[Video description begins] *The fifth code cell displays the following statement: x [6]. The output displays the following error: IndexError: string index out of range.* [Video description ends]

This is very similar to when we work with list. You can't use index values and access elements that are out of range for a particular list, the same thing with a string. There are many similarities between strings and lists but there is a significant difference. Strings in Python are immutable.

[Video description begins] *The sixth code cell displays the following statement: x [0] = 'B'. Following message displays at the bottom-left corner of the screen: Cannot update specific characters in a string.* [Video description ends]

So when assignment statement like the one you see on screen, where you're trying to update the character at index position 0 to be B, this is an error in Python. When you try to perform this assignment, you'll see that Python very clearly tells you that a string object does not support item assignment.

[Video description begins] *A small cross icon displays next to the sixth code cell statement and the output error message is as follows: TypeError: 'str' object does not support item assignment.* [Video description ends]

So remember, you can't update individual characters in the string. You can of course, assign an entirely new string to the variable x that is something we've seen before. Take a look at this assignment statement, this is a completely valid assignment in Python.

[Video description begins] *The seventh code cell displays the following statement: a, b, c, d, e = x. It is followed by the statement: print(a).* [Video description ends]

On the left-hand side of this assignment statement, we have multiple variables, a, b, c, d, and e. And on the right-hand side, we simply have the string x.

[Video description begins] *x = 'World' displays on the right side of the screen.* [Video description ends]

Python will interpret x as a sequence of characters and assign individual characters to each of these variables. If you print out the value of a, it has got a character, W.

[Video description begins] *The output of the seventh code cell is the uppercase alphabet: W.* [Video description ends]

If you print out all of these values, you'll see that each variable has one character from the string World, this is the string stored in x.

[Video description begins] *The eighth code cell displays the following statement: a, b, c, d, e. The output is as follows: ('W ', 'o', 'r', 'l', 'd').* [Video description ends]

These characters have been assigned in sequence to the variables on the left-hand side of the assignment statement. If you're using a tricky assignment statement of this kind, you have to be careful.

[Video description begins] *The ninth code cell displays the following statement: a, b, c, d = x. The alphabets a to d are highlighted in this statement and the alphabet e is highlighted in the seventh code cell.* [Video description ends]

Let's say you only had the variables a, b, c, d on the left-hand side of the assignment statement, but the string x has five characters. You haven't specified enough variables on the left-hand side, and this is an error in Python.

[Video description begins] *The output displays the following error: ValueError: too many values to unpack (expected 4).* [Video description ends]

Python realizes that there are five characters you've only specified four variables and throws an error. This error basically indicates that you were expecting 4 characters in the string X but there were more than 4, so that's an error. An assignment like this is also valid in Python, and it's an important part of the language.

[Video description begins] *The ninth code cell displays the following statement: a, b, _, _, _= x. It is followed by the statement: print(b).* [Video description ends]

[Video description begins] *Two blue-colored arrows point to the alphabets a and b.* [Video description ends]

Notice that we have five place holders on the left, two of these are variables a and b, the others are simply underscores.

[Video description begins] *The three underscores are highlighted.* [Video description ends]

The underscores imply that you know that there are characters at those positions. But you're not interested in those characters, you only want to store the first two characters in a and b. In such an assignment, a will get the value of W, b will get the value o, that is what is printed to screen.

[Video description begins] *The alphabets a and b are highlighted in the code cell. A blue-colored arrow points to the alphabet b in the print statement. The statement x = 'World' displays on the right side of the screen and the alphabets W and o are highlighted. The output displays alphabet o.* [Video description ends]

All of the other characters will be accepted and extracted from the string x but we'll ignore them, because we've assigned them to underscores. A very useful built-in function available in Python is the input function.

[Video description begins] *Now, the three underscores are highlighted in the code cell. The alphabets r, l, and d are highlighted on the right side of the screen.* [Video description ends]

The input function is what you can use to accept some input that the user types in.

[Video description begins] *The 11th code cell displays the following statement: input('How are you?: ').* [Video description ends]

The input function takes in a string that it will display as a message to the user.

[Video description begins] *Following text displays below the Input code cell: How are you?:. Next to it is an empty text field.* [Video description ends]

Once this message is displayed, within this text box you can type in any value which will be interpreted as a string. We've typed in ok, and this is the input accepted by Python. The input built-in function actually returns a value.

[Video description begins] *She types the word: ok. The output below displays the word 'ok' within single quotes. The 12th code cell displays the following statement: place = input('Where are you from?: ').* [Video description ends]

That is, whatever the user has typed in is returned back to the Python program, and you can store this result in a variable.

[Video description begins] *She highlights the input statement when referring to the result and highlights place when referring to the variable.* [Video description ends]

Here we're going to store whatever the user types in in the variable place. So this brings up a text box, within this text box we'll type in New York City.

[Video description begins] *Following text displays below the Input code cell: Where are you from?:. Next to it is an empty text field. She types in, New York City, in the text field. The words New, York, and City are capitalized and separated from each other by a single space.* [Video description ends]

The text that we've typed in, New York City, will be assigned to the variable place. So if you take a look at the value of the place variable, you'll see that it's New York City.

[Video description begins] *The 13th code cell displays the following statement: place. The output below displays the text 'New York City' within single quotes.* [Video description ends]

In this way, you've used the input built-in function to get an input from the user and you've assigned that input to a variable within your program. And now that the value provided by the user is available within a variable, we can perform operations on it. Such as find the length of the input, the number of characters in the input by calling the len built-in function and passing in the place variable.

[Video description begins] *The 14th code cell displays the following statement: len(place). The output displays the value 13. The empty space between the words is also considered in the length.* [Video description ends]

Just like with Python list, strings also support slicing operations.

[Video description begins] *The 15th code cell displays the following statement: place [9:]. Following message displays at the bottom-left corner of the screen: some_string [start_index : end_index]. Below it is the text: start_index: inclusive, end_index: exclusive.* [Video description ends]

For example, you can slice your string 9: and it'll print out all the characters starting from index position 9, all the way through the end of the list, which is just the word City.

[Video description begins] *The output below displays the word 'City' within single quotes.* [Video description ends]

Just like with list, strings also support negative indexing.

[Video description begins] *The 16th code cell displays the following statement: place [-4:]. The place string 'New York City' displays on the right side of the screen.* [Video description ends]

And when you see a negative index within a string square brackets, you start from the end of the string. So -4 at the start index, means start 4 characters from the end of the string and print til the end of the string, which once again, is the word City.

[Video description begins] *The output below displays the word 'City' within single quotes. The word 'City' is also highlighted in the place string.* [Video description ends]

Let's see some more examples of string slicing operations. We want to start at index position 4, and go to position -4, that is till the fourth to last character from the end of the list.

[Video description begins] *The 17th code cell displays the following statement: place [4: -4]. The word York and one space after that are highlighted in the place string. The output below displays the following: 'York '.* [Video description ends]

This gives us the string York followed by the space. Just like with list, string support the step size within the square brackets as well.

[Video description begins] *The 18th code cell displays the following statement: place [0: 8: 2]. Following statement is displayed at the bottom-left corner of the screen: some_string [start_index : end_index :step_size]. Following text displays below: step_size: how to step through the characters in the specified range.* [Video description ends]

So you can see I want to start at index position 0 and go up to the end index 8, not include 8, but I want to access only every second character in the stream, and that is totally possible.

[Video description begins] *The output below displays the string 'NwYr' within single quotes. There is a blue-colored arrow to the alphabet N in the code cell and also to the alphabet N in the place string on the right side of the screen.* [Video description ends]

If you take a look at the result we start at N, that is the first character at index position 0, skip over e, go to w, skip over the space, go to Y, and so on.

[Video description begins] *Three more blue-colored arrows point to the alphabets w, Y, and r in the place string*. [Video description ends]

And once again just like with list, you can slice strings and assign these slices to a variable.

[Video description begins] *The 23rd code cell displays the following statement: city = place [: 8]*. [Video description ends]

For example, the variable City now contains all the characters in your original string up to index position 8.

[Video description begins] *The words New York and one space after that are highlighted in the place string. No output displays as the string has been assigned to the variable city*. [Video description ends]

The character at index position 8 is not included, remember end indices are excluded. Let's say you've assigned a variable that you no longer want used, and you want to free up the memory used by that variable. Remember, variables take up memory on your machine.

[Video description begins] *The 24th code cell displays the following statement: del city*. [Video description ends]

You can always use the del command to free up the memory associated with a variable. If you del city, that variable no longer exists, so if you try to access that variable it'll throw an error.

[Video description begins] *The 25th code cell displays the following statement: city. The following error message displays: NameError: name 'city' is not defined*. [Video description ends]

In the real world you might find yourself working with a lot of data within your Python program. Let's say you have a Python list with a 100,000 elements. If you're no longer using that list, it's often useful to use this del keyword in order to free up the memory associated with that list.

Invoking Functions on Strings

[Video description begins] *Topic title: Invoking Functions on Strings. Your host for this session is Janani Ravi. A jupyter notebook titled StringsAsListsAndStringFunctions displays in the browser window. The Input code cell 26 displays the following statement: print(place.startswith('N'))*. [Video description ends]

Just like lists, you can invoke a number of useful functions on strings as well. For example, strings have a startswith function that allow you to check whether a particular string starts with a certain character. If a particular character or a sequence of characters is a prefix to a string. If your string does start with N, startswith will return true, otherwise, it will return false.

[Video description begins] *There is a blue-colored arrow pointing to the alphabet N in the code cell and also to the alphabet N in the place string on the right side of the screen. The place string displays New York City within single quotes. The string is capitalized. The output below the code cell displays: True*. [Video description ends]

So the return value of startswith is a Boolean value.

[Video description begins] *The code cell 27 displays the following statement: print(place.startswith('n'))*. [Video description ends]

Let's check whether our place starts with the lowercase n. And you'll see that Python correctly returns false. Python knows how to distinguish between uppercase and lowercase characters, just like the startswith function. There's also an endswith function in Python.

[Video description begins] *The code cell 28 displays the following statement: print(place.endswith('y'))*. [Video description ends]

Does place end with y?

[Video description begins] *There is a blue-colored arrow pointing to the alphabet y in the code cell. There is also an arrow pointing to the alphabet y at the end of the word city in the place string. The output below the code cell displays: True*. [Video description ends]

Yes indeed, it does, and the case matches as well. startswith and endswith can be used with strings as well. Does place end with the string city?

[Video description begins] *The code cell 29 displays the following statement: print(place.endswith('City')). There is a blue-colored arrow pointing to the word City in the code cell and the word City in the place string is highlighted*. [Video description ends]

Yes it does, which is why the return value of this function is true.

[Video description begins] *The output below the code cell displays: True*. [Video description ends]

There are other helpful functions available with strings as well, such as a count function, which will count the number of occurrences of a particular character or string.

[Video description begins] *The code cell 30 displays the following statement: place.count('y')*. [Video description ends]

The number of occurrences of the character y is exactly 1 in New York City.

[Video description begins] *She points to the alphabet y at the end of the place string. The output displays the value 1*. [Video description ends]

Observe once again that the case matters here. The y of York is an uppercase, and that is not included in this count. Let's use another useful string function to convert the current string to lower case. place.lower() will lower case the string stored in place.

[Video description begins] *The code cell 31 displays the following statement: lower_place = place.lower(). It is followed by the statement: lower_place*. [Video description ends]

And the return value will be the original string, all lower case, which will be assigned to lower_place variable.

[Video description begins] *The output displays the following string in all lowercase alphabets: 'new york city'. The place string with initial uppercase alphabets also displays on the right side of the screen*. [Video description ends]

Just like the lower function, you have an upper function on a string. place.upper will convert the string stored in place to upper case and we have stored the result in upper_place.

[Video description begins] *The code cell 32 displays the following statement: upper_place = place.upper(). It is followed by the statement: upper_place*. [Video description ends]

[Video description begins] *The output displays the following string in all uppercase alphabets: 'NEW YORK CITY'*. [Video description ends]

Now let's count the number of occurrences of the uppercase y in the string New York City where all of the characters are in uppercase. And you see that the count is 2, we have two y's. 1 from York and 1 from the end of City.

[Video description begins] *The code cell 33 displays the following statement: upper_place.count('Y'). The upper_place string displays on the right side of the screen and displays the text NEW YORK CITY in uppercase.*

Two arrows point to the alphabets Y in the words YORK and CITY. The output displays the value 2. [Video description ends]

If you want to check whether the value stored in two variables are equal, you'll use the double equal to operator.

[Video description begins] *The code cell 34 displays the following statement: lower_place == upper_place.* [Video description ends]

This is a conditional operator which performs a check for equality. So essentially, this operator is going to check whether the string stored in the lower_place variable is equal to the string stored in the upper_place variable.

[Video description begins] *Following text displays at the bottom left of the screen: Different from assignment which uses just "=".* [Video description ends]

If indeed it is, the return value will be true, otherwise it will be false. lower_place is obviously not equal to upper_place, which is why the conditional operator returns false.

[Video description begins] *The lower_place and upper_place variables display on the right side of the screen along with their values. The lower_place has the string 'new york city' in all lowercase and the upper_place has the string 'NEW YORK CITY' in all uppercase. There is a not equal to sign between them. The output displays False.* [Video description ends]

Is the string value in the place variable equal to the string value in the upper_place variable? This is once again, false. And that is what the conditional operator returns.

[Video description begins] *The code cell 35 displays the following statement: place == upper_place. The place and upper_place variables display on the right side of the screen along with their values. The place has the capitalized string 'New York City' and the upper_place has the string 'NEW YORK CITY' in all uppercase. There is a not equal to sign between them. The output displays False.* [Video description ends]

There are other useful functions available on strings that you can invoke, such as the find function, which will return the index position where you can find the character C.

[Video description begins] *The code cell 36 displays the following statement: place.find('C'). The place variable displays on the right side of the screen along with the capitalized string 'New York City'. There is an arrow pointing to the alphabet C of the word City. The output displays the value 9.* [Video description ends]

And in the string New York City, this happens to be the index position 9. Let's try the find function once again. This time we pass in the string York as an input argument to find. We want to find the index position where York occurs in our string, and this happens to be index position 4.

[Video description begins] *The code cell 37 displays the following statement: place.find('York'). The place variable displays on the right side of the screen along with the capitalized string 'New York City'. The word York is highlighted. The output displays the value 4.* [Video description ends]

Strings in Python also support another function called index, which returns the index position for the string or character that you pass in. Place.index "Y" will return 4.

[Video description begins] *The code cell 38 displays the following statement: place.index('Y'). There is an arrow pointing to the alphabet Y of the word York. The output displays the value 4.* [Video description ends]

The character Y occurs at index position 4 in our string New York City, but index here, if you try to pass in a character that does not exist in a string, you'll get an error. It says the substring was not found.

[Video description begins] *The code cell 39 displays the following statement: place.index('c'). The output displays the following error: ValueError: substring not found.* [Video description ends]

So, the substring c, the c in lower case was not found in the place string. The original string New York City only contains the letter c in uppercase. When you try to find the index value of a non-existing string using the find function, it'll behave a little differently. It'll simply return -1 to indicate that the string that you were looking for within the place variable was not found.

[Video description begins] *The code cell 40 displays the following statement: place.find('c'). The output displays the value -1.* [Video description ends]

So the index and the find functions on strings perform very similar operations, except that one throws an error when the string is not found, that is the index function. And the find function does not throw an error, but instead returns -1. Let's take a look at some more useful functions available on strings, such as the split function.

[Video description begins] *The code cell 41 displays the following statement: split_place = place.split('k'). Following statement displays below: split_place.* [Video description ends]

The split function allows you to split a string on a character. Here, we've passed in k as a character. And you'll find that the return value is a list of strings which have been split where the character k occurs in the original string.

[Video description begins] *The output displays the following: ['New Yor', ' City']. She highlights the alphabet k in the place string on the screen.* [Video description ends]

The result split place is a list. The first element in this list comprised of the string before the character key in the original string. And the second element of the list is City, which comprise of all the characters which occur after the character k in our original string.

[Video description begins] *She highlights 'New Yor' before the alphabet k in the place string. She then highlights the characters after k, which is the word City preceded by an empty space.* [Video description ends]

This split function is often used to extract words from a string. Let's count the number of spaces in our original string.

[Video description begins] *The code cell 42 displays the following statement: place.count(' '). There is an empty space between the single quotes. There are two arrows pointing to the empty spaces between the words New York and York City respectively. The output displays the value 2.* [Video description ends]

There are two spaces in our string New York City, and let's split on the space. This is the operation that will allow us to extract the words in a string. Take a look at split_place, you now get New, York and City.

[Video description begins] *The code cell 43 displays the following statement: split_place = place.split(' '). There is an empty space between the single quotes. The code cell 44 displays the following statement: split_place. All three words in the place string are highlighted separately on the right side of the screen. The output displays the following: ['New', 'York' , 'City'].* [Video description ends]

So when you split on the space, words from your original string are present in the result. The spaces have been eliminated, you only get the words. And if you see the length of the split_place list, it'll give you 3, corresponding to the 3 words that are in that particular list.

[Video description begins] *The code cell 45 displays the following statement: len(split_place). The output displays the value 3.* [Video description ends]

Just like the split function, the reverse of the split function is the join function, available on strings.

[Video description begins] *The code cell 46 displays the following statement: join_char = ' , '. It is followed by the statement join_char.join(split_place).* [Video description ends]

Here I've initialize a string called join_char. This is a string with just one character, the comma. And I've invoked the join function on this join_char string, join_char.join, and I pass in a list of strings. That is the split_place. You can see that the result here is all of the strings in this split_place list New, York, and City were the words within it, joined using the comma. Every word is separated using the join character, the comma.

[Video description begins] *The output displays the three words New York City enclosed within single quotes and separated by commas*. [Video description ends]

Let's try the same thing again, but this time we'll have the space as the join character, join character 1.join(split_place) will give us New York City, where New, York and city are separated by spaces.

[Video description begins] *The code cell 47 displays the following statement: join_char1 = ' '. There is an empty space between the single quotes. It is followed by the statement join_char1.join(split_place). The output displays the three words New York City enclosed within single quotes and each word is separated from the other by a space*. [Video description ends]

You'll find that the join function is simply invoked on a string in quotes. So we have the pipe operator here, the vertical line.

[Video description begins] *The code cell 48 displays the following statement: "|".join(split_place). The output displays the three words New York City enclosed within single quotes and separated by a pipe symbol*. [Video description ends]

That is string we call.join and pass in split_place and you get New York City, where the words are separated using the pipe character.

Perform Slicing Operations on Strings

[Video description begins] *Topic title: Perform Slicing Operations on Strings. Your host for this session is Janani Ravi*. [Video description ends]

In this video, we'll see some interesting stuff that we can do with strings in Python. A string in Python is essentially a sequence of characters. And once you have a sequence, you can index into the string to access specific ranges of characters in the string. What does that mean?

[Video description begins] *A jupyter notebook titled StringsAsSequenceOfCharacters displays in the browser window. The Input code cell 1 displays the following statement: my_string = "Hello Python World". Below it is the statement: print(my_string [0:18])*. [Video description ends]

Observed this my_string, which contains the value, Hello Python World. And observe how we use square brackets to index into the string.

[Video description begins] *She uses two blue-colored arrows to point to the square brackets in the code cell*. [Video description ends]

Here, the square brackets allows you to specify the range of characters that you want to access from the string. Starting from the start _index, then you have a colon and then the end_index. The first character of a string is always at index position 0, which is why we've said we're interested in all characters starting from index position 0.

[Video description begins] *Following message displays at the bottom-left corner of the screen: string_name [start_index :end_index]. Below it is the text: The first character of a string is at index position 0. Two blue-colored arrows point to the character H and the value 0 respectively. The term start_index is highlighted at the bottom of the screen*. [Video description ends]

That is the first character of the string, all the way to index position 18 the character which is at position 18. The start index here is inclusive, which means the character at index position 0 will be included.

[Video description begins] *A blue-colored arrow points to the value 18. The term end_index is highlighted at the bottom of the screen.* [Video description ends]

The end index is exclusive. The character at index position 18 will not be included in the result. Now it so happens that 18 is the length of this string, which is why this gives you the string as a whole, Hello Python World is the result. But what if you want to access only a substring from the original string? Here the start index is 0 that is the first character of the string.

[Video description begins] *The 2nd code cell displays the following statement: print (my_string [0:6]).* [Video description ends]

The end index is 6. Remember the end index is exclusive which will give you all of the characters from index 0 all the way to index 5. That is the first six characters the word Hello.

[Video description begins] *Two blue-colored arrows point to the values 0 and 6 respectively.* [Video description ends]

The character o which is the sixth character is at index position 5. Let's try indexing into our stream once again. This time we start with the character at index position 6 and go all the way up to position 18.

[Video description begins] *The 3rd code cell displays the following statement: print(my_string [6:18]). A blue-colored arrow points to the value 6. Another blue-colored arrow points to the alphabet P of the word Python in the code cell 1.* [Video description ends]

Remember, 18 marks the end of our string. So this will give us the result Python World. The letter P is at index position 6, that's where we start.

[Video description begins] *The output displays the following: Python World.* [Video description ends]

As long as the index position is within the range, Python will do the right thing.

[Video description begins] *The 4th code cell displays the following statement: print(my_string [6:12]). The text, my_string = "Hello Python World", displays on the right side of the screen.* [Video description ends]

Here, you're interested in all characters starting from index position 6 going all the way, but not including index position 12.

[Video description begins] *The word "Python" is highlighted on the right side of the screen and the output displays: Python.* [Video description ends]

This is just the word Python. Python has some very powerful techniques to access specific characters in a string.

[Video description begins] *The 5th code cell displays the following statement: my_string = "Hello Python World". Below it is the statement: print(my_string [: :2]). Following statement is displayed at the bottom-left corner of the screen: [start_index : end_index :step_size]. Following text displays below: step_size - how many characters to skip in the specified range.* [Video description ends]

For example, it's not just the start_index and end_index that you specified within square brackets, you can also specify a step_size.

[Video description begins] *She points to the value 2 and the text: step_size is highlighted at the bottom.* [Video description ends]

The step_size basically indicates how many characters you want to skip while you're accessing the characters within the range specified. Let's take a look at the example that you see here on screen. Here we access my_string, which is the string Hello Python World. Notice that the start index and end index are completely empty, which means we want to access the entire string.

When we omit the start index and end index, it refers to the entire string. Here, we've only specified a step size of 2. Now in order to explain this, I'm going to display the result to you and show you how we got the result.

[Video description begins] *The output displays the following characters: HloPt ool.* [Video description ends]

Since you omitted the start index within square brackets, we start at the very beginning of the string at the character H.

[Video description begins] *A blue-colored arrow points to the uppercase H in the output. Another blue colored arrow points to the character H in the string Hello Python World, which is displayed on the right side of the screen.* [Video description ends]

The character H is printed to screen in the result. We then print every second character after that. So after the H, we skip over the e, print the first l skip over the second l, print the o, skip over the blank space print the P, skip over the y, print the t and so on.

[Video description begins] *The screen displays blue-colored arrows next to the following characters: loPtool.* [Video description ends]

The step size specifies how we step over the characters within the input range. Now we haven't encountered the step size before. That's because the default value for step size is 1.

[Video description begins] *The 6th code cell displays the following statement: my_string = "Hello Python World". Below it is the statement: print(my_string [0:18 :1]).* [Video description ends]

When you use a step size of 1, it simply prints out the entire character in the range, which is in the range 0 to 18, which for our case is the string as a whole. Here is one other example.

[Video description begins] *The output displays the following: Hello Python World.* [Video description ends]

We start at index position 0 go all the way to 18 and the step size is 2.

[Video description begins] *The 7th code cell displays the following statement: my_string = "Hello Python World". Below it is the statement: print(my_string [0:18 :2]).* [Video description ends]

Here is every second character in the entire input string. Python is a powerful and complex language.

[Video description begins] *The output displays the following characters: HloPt ool.* [Video description ends]

And in fact, it allows you to step through your characters in the reverse direction as well when you specify a negative step size.

[Video description begins] *The 8th code cell displays the following statement: my_string = "Hello Python World". Below it is the statement: print(my_string [18:0 :-2]).* [Video description ends]

Here what we're essentially saying is start at the very end of the string at position 18. Go up to the very beginning of the string up to position 0 and step backward using a step size of -2. And this exactly what you see here in the result. We start at the very last character, which is the character d. We then skip over l and then print out r, skip over o, print out W, skip over the space, print out n, and so on allowing you to step over a string backwards means that if you want to reverse a string, you can do so very easily.

[Video description begins] *The output displays the following characters: drWnhy le. A blue-colored arrow points to the character d in the output. Another blue colored arrow points to the character d in the string Hello Python World, which is displayed on the right side of the screen,* [Video description ends]

Here we start with the character at index position 4.

[Video description begins] *The 9th code cell displays the following statement: my_string [4::-1].* [Video description ends]

We haven't specified an end index here, and the end index will be determined by our step size.

[Video description begins] *A blue-colored arrow points to the value 4 and the character o at the end of the word Hello is highlighted on the right side of the screen.* [Video description ends]

Our step size here is -1, indicating that we want to walk through the string backward. And as the result shows us here, we start with the character o, which is at index position 4, and go all the way to the beginning of the string, stepping backward.

[Video description begins] *The output displays the following characters within single quotes: 'olleH'.* [Video description ends]

If you want to reverse a string in Python, all you need to do is to omit the start and end indices and specify -1 as your step size.

[Video description begins] *The 10th code cell displays the following statement: my_string = "Hello Python World". Below it is the statement: print(my_string [: :-1]).* [Video description ends]

This will walk through every character of your string backward, thanks to the -1, starting at the very end of your string so here is your entire string in reverse. That is the result. A string in Python is actually a class.

[Video description begins] *The output displays the following characters: dlroW nohtyP olleH.* [Video description ends]

You haven't studied classes yet, you don't need to worry about it. But you should know that classes support functions that can be invoked on those classes, which means that you can invoke certain functions on your strings.

[Video description begins] *The 11th code cell displays the following statement: my_string = "Hello Python World". Below it is the statement: some_string = my_string.split(). Below it is the statement: some_string.* [Video description ends]

Here, the variable my_string contains the string Hello Python World. And we've used the dot operator and then invoke the split function on my_string. The dot over here basically implies that the split function is invoked on my_string. And it will apply to the contents of my_string that is Hello Python World is what will be split, and it will be split into its individual words by default.

[Video description begins] *The output displays the following: ['Hello','Python','World'].* [Video description ends]

Notice that my_string.split returns a list of words enclosed in square brackets. The presence of square brackets indicates that this is a Python list. Python has split the original string into individual words and stored the results in some string. The original string remains unchanged. A Python list is basically a list of elements of any kind enclosed within square brackets, and we'll talk a lot more about list later on.

There are other functions that you can invoke on strings as well, such as the upper functions. So we have my_string hello python world all in lowercase calling the upper function will return a result where the original string is now in all uppercase. Instead of using a variable, [Video description begins] *The 12th code cell displays the following statement: my_string = "hello python world". All characters are in lowercase. Below it is the statement: my_string.upper(). The output displays the HELLO PYTHON WORLD string within quotes and it has all uppercase characters.* [Video description ends]

you can also invoke functions on a string by specifying the string in quotes. Your HELLO PYTHON WORLD all in uppercase is a quoted string we call the dot is upper function on it to see whether all of the string is in uppercase.

[Video description begins] *The 13th code cell displays the following statement: "HELLO PYTHON WORLD".isupper(). The string HELLO PYTHON WORLD is in all uppercase.* [Video description ends]

The return value is true because yes, indeed it is. Let's initialize the my_string variable again to HELLO PYTHON WORLD in all uppercase.

[Video description begins] *The 14th code cell displays the following statement: my_string = "HELLO PYTHON WORLD". The string HELLO PYTHON WORLD is in all uppercase. Below it is the statement: my_string.* [Video description ends]

And let's call the lower function on it. Just like the upper function, my_string.lower will take the value of the original string, convert it to lowercase, and return that as the result here.

[Video description begins] *The output displays the HELLO PYTHON WORLD string within quotes and it has the all uppercase characters. The 15th code cell displays the following statement: my_string.lower().* [Video description ends]

Just hello python world all in lowercase.

[Video description begins] *The output displays the complete hello python world string in lowercase and is within quotes.* [Video description ends]

Introducing Tuples

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

Another complex data type that is commonly used in Python is the tuple. The tuple is very similar to list in many ways but they're also different. In this particular demo, we'll explore how tuples are similar to list.

[Video description begins] *A jupyter notebook titled TuplesAndSimilaritiesBetweenListAndTuple displays in the browser window.* [Video description ends]

Exactly like list in Python, tuples are an ordered collection of elements. The order of fields or elements within a tuple matters.

[Video description begins] *The Input code cell 1 displays the following statement: my_tuple = (). Below it is the statement: my_tuple.* [Video description ends]

A tuple is initialized using rounded brackets, as opposed to square brackets for list. Here we have an empty tuple that we've initialized to the my_tuple variable. You can see that this tuple is empty.

[Video description begins] *The output displays: (). The 2nd code cell displays the following statement: type(my_tuple).* [Video description ends]

Let's see the type of my_tuple. You'll see that Python recognizes this as the complex data type tuple.

[Video description begins] *The output displays: tuple. The 3rd code cell displays the following statement: print(type(my_tuple))*. [Video description ends]

Let's print out this type and you'll find that the result is a little different by concess this is of class tuple. This is because these complex types under the hood are actually classes in Python.

[Video description begins] *The output displays the following: <class 'tuple'>*. [Video description ends]

Only thing that you need to know about classes at this point in time is that they serve as a template for what a particular type looks like. And the functions you can invoke on that type strings, list, tuples, dictionaries and sets are all classes under the hood. Lets initialize a tuple with three numeric values 1, 2, and 3.

[Video description begins] *The 4th code cell displays the following statement: int_tuple = (1, 2, 3). Below it is the statement: print(int_tuple)*. [Video description ends]

Notice that this initialization is only different from that of a list in that the three values are enclosed within rounded brackets as opposed to square bracket.

[Video description begins] *The output displays as follows: (1, 2, 3)*. [Video description ends]

Just like a tuple of numeric types, you can have a tuple of strings. Here we have a string tuple with the strings Hello and Python within the tuple.

[Video description begins] *The 5th code cell displays the following statement: str_tuple = ("Hello", "Python"). Below it is the statement: print(str_tuple). The output displays as follows: ('Hello', 'Python')*. [Video description ends]

The print function in Python can take in two tuples, and it'll print out each of these tuples within rounded brackets.

[Video description begins] *The 6th code cell displays the following statement: print(int_tuple, str_tuple). The output displays as follows: (1, 2, 3) ('Hello', 'Python')*. [Video description ends]

So 1, 2, 3 is within one set of rounded brackets. That is one tuple. And Hello Python is within another set of rounded brackets. Just like in the case of strings, you can have a tuple which has data of different types within it. Here, the combined tuple we've formed by adding together the elements of the int_tuple and the str_tuple.

[Video description begins] *The 7th code cell displays the following statement: combined_tuple = int_tuple + str_tuple. Below it is the statement: print(combined_tuple)*. [Video description ends]

And you can see that the combined tuple contains the numbers 1, 2, 3, as well as the strings Hello and Python.

[Video description begins] *The output displays as follows: (1, 2, 3, 'Hello', 'Python')*. [Video description ends]

Python also supports these operations where you can multiply a particular tuple.

[Video description begins] *The 9th code cell displays the following statement: print(str_tuple * 3). Contents of the 5th code cell and its output display on the right side of the screen*. [Video description ends]

And the result is a tuple where the elements from the original tuple have been repeated thrice. So you multiplied string tuple by three, Hello Python is present thrice in the result in tuple.

[Video description begins] *The output displays as follows: ('Hello', 'Python', 'Hello', 'Python', 'Hello', 'Python'). She labels the three occurrences of Hello and Python as 1, 2, and 3 respectively in the output string*. [Video description ends]

Here is a tuple of elements, where the elements are of different data types, an integer, a floating point, and a streak here is the mixed tuple.

[Video description begins] *The 10th code cell displays the following statement: mixed_tuple = (1, "Hello", 3.4). Below it is the statement: print(mixed_tuple). The output displays as follows: (1, 'Hello', 3.4).* [Video description ends]

Now in Python, any time you assign a value to a variable where the value is separated by commas, Python automatically interprets this as a tuple. So even though you left out the rounded brackets, when you assigned the my_tuple variable, Python correctly realize that my_tuple should be a tuple.

[Video description begins] *The 11th code cell displays the following statement: my_tuple = 1, "Hello", 3.4. The opening and closing brackets are missing in this statement. Below it is the statement: print(my_tuple). The output displays as follows: (1, 'Hello', 3.4).* [Video description ends]

When talking about what a list is made up of, we often talk about the elements of a list. Fields are what we use to refer to the elements in a tuple. Let's say you wanted to access the individual fields in a tuple and assign these field values to variables.

[Video description begins] *The 12th code cell displays the following statement: a, b, c= my_tuple. Below it, are the following statements on separate lines: print(a), print(b), and print(c).* [Video description ends]

Here is an assignment statement that is valid in Python. A, b, c = my_tuple will assign the first field of your tuple to the variable a, the second field to the variable b, and the third field to the variable c.

[Video description begins] *The following output displays on separate lines: 1, Hello, and 3.4.* [Video description ends]

Just like you can have a list which includes another list as its element that is nested list. You can have nested tuples as well.

[Video description begins] *The 13th code cell displays the following statement: nested_tuple = (1, 2, 3, (4, 5, 6)). Below it is the statement: print(nested_tuple).* [Video description ends]

Here observe that 4, 5, and 6 are a nested tuple within the outer tuple.

[Video description begins] *She highlights the tuple, which contains the values 4, 5, and 6 within brackets.* [Video description ends]

The first three fields of this nested tuple are the numbers 1, 2, and 3. And the fourth field is the tuple which contains the fields 4, 5, and 6.

[Video description begins] *The following output displays: (1, 2, 3, (4, 5, 6)).* [Video description ends]

Tuples can contain not just nested tuples but also nested lists. So you can have a list within a tuple, for example, the list here, 8 ,4 ,6 is within the outer tuple, which we've assigned to the variable mixed tuple.

[Video description begins] *The 14th code cell displays the following statement: mixed_tuple = ("mouse", [8, 4, 6], (1, 2, 3)). Below it is the statement: print(mixed_tuple).* [Video description ends]

The result here is a mixed tuple, which has the string as its first field, a list as its second field, and a nested tuple as its third field.

[Video description begins] *The following output displays: ('mouse', [8, 4, 6], (1, 2, 3)). She points to the string 'mouse' as the first field; the list containing 8, 4, and 6 as the second field; and the nested tuple containing 1, 2, and 3 as the third field.* [Video description ends]

Let's set up a tuple of characters here, which contain the characters of the term permit.

[Video description begins] *The 15th code cell displays the following statement: my_tuple = ('p', 'e', 'r', 'm', 'i', 't'). Below it is the statement: my_tuple. The output displays all the characters in single quotes separated by commas and enclosed within the simple bracket.* [Video description ends]

Accessing the fields of a tuple is very similar to how you would access the elements in a list. My_tuple of 0 will access the first field, which contains the character p.

[Video description begins] *The 16th code cell displays the following statement: print(my_tuple [0]). The output is the character p.* [Video description ends]

Let's see another example, my_tuple of 5 will access the field at index position 5, which is t.

[Video description begins] *The 17th code cell displays the following statement: print(my_tuple [5]). The output displays the character t.* [Video description ends]

Just like with list, tuple fields start at index position 0. If you try to access a field that is out of range for a particular tuple, such as index value 6.

[Video description begins] *The 18th code cell displays the following statement: print(my_tuple [6]). The output displays the following message: IndexError: tuple index out of range.* [Video description ends]

You'll find that we'll get an error which says tuple index out of range. Let's access the field at index position one of our mixed tuple.

[Video description begins] *The 23rd code cell displays the following statement: print(mixed_tuple [1]). The mixed_tuple displays to the right side of the screen. The output displays the following: [8, 4, 6].* [Video description ends]

This is the list, which contain the elements 8, 4, and 6. Now if you want to access a particular element of this list which is contained within the tuple, you can use two square brackets.

[Video description begins] *The 24th code cell displays the following statement: print(mixed_tuple [1] [0]).* [Video description ends]

The first square bracket is used to access the field within the outer tuple that is our mixed tuple.

[Video description begins] *She points to the square bracket containing the value 1. The field containing values 8, 4, and 6, is highlighted in the mixed_tuple, which is displayed on the right side of the screen.* [Video description ends]

And the second square bracket is used to access the element within the nested list.

[Video description begins] *She points to the square bracket containing the value 0. She also points to the digit 8 in the highlighted field. The output displays the value 8.* [Video description ends]

The element at index 2 within our mixed tuple is our nested tuple containing the fields 1, 2, and 3.

[Video description begins] *The 25th code cell displays the following statement: nested_tuple = mixed_tuple [2]. Below it is the statement: print(nested_tuple). The output displays the following: (1, 2, 3).* [Video description ends]

If you want to access a field of this nested tuple, you can use the two square brackets once again.

[Video description begins] *The 26th code cell displays the following statement: print(mixed_tuple [2] [2]). The output displays the value 3.* [Video description ends]

We access the nested tuple at index position 2. And then access the field within the nested tuple at index position 2, the value is 3. Just like with list, tuples also support slicing operations.

[Video description begins] *The 27th code cell displays the following statement: my_tuple = ('E', 'l', 'e', 'p', 'h', 'a', 'n', 't'). Below it is the statement: print(my_tuple [1:]).* [Video description ends]

Here is a tuple of characters of the word elephant, and I want to access all of the characters starting from index position 1 onwards. And I get all of the characters, starting from l, which is the field at index position one in our tuple. Performing slicing operations on a list gives you a list performing slicing operations on a tuple gives you a tuple. And you can specify ranges just like you would in a list. I want all of the characters from two up to but not including 4, that gives me the characters e and p.

[Video description begins] *The 28th code cell displays the following statement: print(my_tuple [2:4]). Two arrows point to the digits 2 and 4. The output displays the characters e and p in single quotes separated by commas and enclosed within the simple brackets.* [Video description ends]

In addition to start and end indices, you can specify step size as well. Here I want to access every second character from the entire list and that's what I get here in the result. Tuples also support negative index values, so if you access the field at index position -1.

[Video description begins] *The 29th code cell displays the following statement: print(my_tuple [::2]). The output displays the following characters in single quotes separated by commas and enclosed within the simple brackets: E, e, h, and n.* [Video description ends]

That will give you the last character in our tuple, the last field in our tuple, the character t.

[Video description begins] *The 30th code cell displays the following statement: print(my_tuple [-1]). The output is the character t.* [Video description ends]

We'll now initialize another tuple, my_tuple containing the characters of the word Hello.

[Video description begins] *The 33rd code cell displays the following statement: my_tuple = ('H', 'e', 'l', 'l', 'o'). Below it is the statement: my_tuple. The output displays all the characters of the word Hello.* [Video description ends]

Just like lists or strings, tuples also support functions that you can invoke on them such as the index function, which returns the index position of the input argument that you pass in. Here I want the index position of the character H, that is 0.

[Video description begins] *The 34th code cell displays the following statement: print(my_tuple.index('H')). The character H is in uppercase. The output displays the value 0.* [Video description ends]

Python also supports the special keyword, the in keyword which you can use to check whether a particular element or field is present in a tuple or a list.

[Video description begins] *The 37th code cell displays the following statement: print('e' in my_tuple). The character e is in lowercase. The output displays the value True.* [Video description ends]

E in my_tuple will return True because my_tuple indeed contains the character E. Let's check whether the character t is in my_tuple, t in my_tuple will return false because the only characters are H, e, l, l, o.

[Video description begins] *The 38th code cell displays the following statement: print('t' in my_tuple). The output displays the value False.* [Video description ends]

The character t is not present. Let's check this once again, let's check whether e is not in my_tuple.

[Video description begins] *The 39th code cell displays the following statement: print('e' not in my_tuple). The character e is in lowercase.* [Video description ends]

Not in is also a keyword and also, observe how these Python keywords are basically just English. E is not in my_tuple, that's False because e is indeed present in my_tuple, in the characters of the word Hello. Let's say I have an int tuple containing the integers 1, 4, 9, and 9. Just like with a list, I can pass in this tuple to a built in function such as sum, to get the sum of all values of the tuple, which in this case is 23.

[Video description begins] *The 48th code cell displays the following statement: int_tuple = (1, 4, 9, 9). Below it is the statement: sum(int_tuple). The output is the value 23.* [Video description ends]

Let's consider this int_tuple, 1, 4, 9, 9, I can convert this to a list as well, simply pass in the int_tuple to our list function.

[Video description begins] *The 49th code cell displays the following statement: int_tuple = (1, 4, 9, 9). Below it is the statement: list(int_tuple).* [Video description ends]

And this will convert our tuple to a list, and you can see that the result is a list because of the square brackets. We can take the same int_tuple and convert it to a set.

[Video description begins] *The output displays the numbers 1, 4, 9, and 9, separated by commas and enclosed within the square brackets.* [Video description ends]

Remember, a set cannot contain duplicate values, we can see the result is a set, as specified by the curly brackets. And you can see that the duplicate value 9 has been removed from our original tuple. The set contains just 1,4,9. No duplicate values are allowed in the set.

[Video description begins] *The 50th code cell displays the following statement: int_tuple = (1, 4, 9, 9). Below it is the statement: set(int_tuple). The output displays the following: {1, 4, 9}.* [Video description ends]

Understanding Tuple Immutability

[Video description begins] *Topic title: Understanding Tuple Immutability. Your host for this session is Janani Ravi. A jupyter notebook titled TuplesAndDifferencesBetweenListAndTuple displays in the browser window.* [Video description ends]

If tuples and strings are exactly the same, why do we need both? That seems strange. Well it's not strange because tuples and strings are actually different. And the significant difference is the fact that tuples are immutable. Tuples once created cannot be updated. So you'll tend to prefer tuples over lists if you want an immutable collection. Let's see an example of immutable tuples, here is a tuple here of numbers.

[Video description begins] *The 1st code cell displays the following statement: immutable_tuple_1 = (0, 1, 2, 3). Below it is the statement: immutable_tuple_1. The output displays: (0, 1, 2, 3).* [Video description ends]

This tuple cannot be assigned values, so let's say you want to try and use the square brackets to assign a new value to a particular tuple field.

[Video description begins] *The 2nd code cell displays the following statement: immutable_tuple_1 [0] = 4. Below it is the statement: print(immutable_tuple).* [Video description ends]

This will result in an error. We are trying to update the number at index position 0 for this tuple, and you'll see that Python clearly tells you that the tuple object does not support item assignment.

[Video description begins] *The output contains the following message: TypeError: 'tuple' object does not support item assignment*. [Video description ends]

Let's say you have a nested tuple, a tuple of tuples where one of the fields itself is a tuple, such as the tuple containing 6, 5.

[Video description begins] *The 3rd code cell displays the following statement: immutable_tuple_2 = (4, 2, 3, (6, 5)). Below it is the statement: immutable_tuple_2. The output displays the following: (4, 2, 3, (6, 5))*. [Video description ends]

You can access individual fields in a tuple using the square brackets, such as we've accessed the field at index value 0.

[Video description begins] *The 4th code cell displays the following statement: immutable_tuple_2 [0]. The output is the value 4*. [Video description ends]

But if you try to update this particular field in a tuple, let's say assign it a value 9, that is an error.

[Video description begins] *The 5th code cell displays the following statement: immutable_tuple_2 [0] = 9. The output displays the following message: TypeError: 'tuple' object does not support item assignment*. [Video description ends]

Tuples are immutable and this is how this significantly different from list. Say you have a list field within a tuple, we have an immutable tuple here.

[Video description begins] *The 6th code cell displays the following statement: immutable_tuple_2 = (4, 2, 3, [6, 5]). Below it is the statement: immutable_tuple_2*. [Video description ends]

And within that the last field in the tuple is a list containing the elements 6, 5. Now, this tuple itself is immutable, but the list within it is not.

[Video description begins] *The output displays: (4, 2, 3, [6, 5])*. [Video description ends]

Let's see what I mean here. At index 3 within this immutable tuple is our list containing the elements 6, 5.

[Video description begins] *The 7th code cell displays the following statement: immutable_tuple_2 [3]. The output displays: [6, 5]*. [Video description ends]

Let's say we were to use the double square brackets. Here, we first access the list within the tuple and try and assign a value to an element of that list.

[Video description begins] *The 8th code cell displays the following statement: immutable_tuple_2 [3] [1] = 100. Below it is the statement: print(immutable_tuple_2)*. [Video description ends]

We're accessing the list at index 3, and we are setting the first index of that list to value 100, this is allowed.

[Video description begins] *She points to the indexes 3 and 1 and then highlights the value 100*. [Video description ends]

Observe how the list within the tuple has been updated and now contains the elements 6,100. The tuple itself is immutable, the list within the tuple is not.

[Video description begins] *The output displays: (4, 2, 3, [6, 100])*. [Video description ends]

I'll now create a new tuple containing the characters of the word Python, so you can see that my_tuple is a tuple of characters.

[Video description begins] *The 9th code cell displays the following statement: my_tuple = ('p', 'y', 't', 'h', 'o', 'n'). Below it is the statement: my_tuple. The output displays all the characters in single quotes separated by commas and enclosed within the simple bracket.* [Video description ends]

This tuple is immutable and you will find that if you try to delete a particular character within this tuple, the one at index position 3.

[Video description begins] *The 10th code cell displays the following statement: del my_tuple [3]. The output contains the following message: TypeError: 'tuple' object does not support item deletion.* [Video description ends]

This is also an error, the tuple object does not support item deletion. A tuple once created cannot have its fields updated, you can't delete its fields as well, but you can delete the tuple as a whole. You can use the del keyword and free up the memory corresponding to my_tuple.

[Video description begins] *The 11th code cell displays the following statement: del my_tuple.* [Video description ends]

Now if you try to access the my_tuple variable, Python will throw an error, my_tuple is no longer defined since you've deleted it.

[Video description begins] *The 12th code cell displays the following statement: print(my_tuple). The output contains the following message: NameError: name 'my_tuple' is not defined.* [Video description ends]

Now that we've studied the important difference between tuples and lists, let's study some interesting operations we can perform with tuples. Here we have two tuples, tuple_a and tuple_b.

[Video description begins] *The 13th code cell displays the following statements: tuple_a = (1, 2, 3, 4, 5). Below it is the statement: tuple_b = ('a', 'b', 'c', 'd', 'e').* [Video description ends]

Tuple_a is a tuple of numbers and tuple_b is a tuple of characters. I'm going to introduce you to the zip function in Python. This is extremely useful and often used when you're using data science libraries.

[Video description begins] *The 14th code cell displays the following statement: zipped = zip(tuple_a, tuple_b). Below it is the statement: print(zipped).* [Video description ends]

The zip function is so called because it essentially acts as a zipper would on your clothes. It basically makes pairs of every element in tuple_a with the corresponding element in tuple_b. The result of the built in zip function is simply an object. You can't really tell what the structure of this object is.

[Video description begins] *The output displays the following: <zip object at 0x10eb20b08>* [Video description ends]

In its simplest form you can think of the zip data structure as a collection of tuples. Every tuple will have one field from tuple_a and one field from tuple_b. These are the original tuples that you zipped up together.

[Video description begins] *Following message displays at the bottom left of the screen: collection of (1, ' a') (2, ' b') (3, ' c') (4, ' d') (5, ' e').* [Video description ends]

Now what exactly this collection is depends on how you use the zipped result.

[Video description begins] *The 15th code cell displays the following statement: result = tuple(zipped). Below it is the statement: print(result).* [Video description ends]

So if you pass in the zipped result to a tuple function, you'll get a tuple of tuples. Here in this example we've converted the zipped result to a tuple. The zipped result can be converted to a tuple, it can be converted to a list, a dictionary or even a set, any complex data type that Python supports. Once we convert the zipped object to a tuple, we'll store it in the result variable, and here is what the result variable looks like. The result here is a tuple of tuples.

[Video description begins] *The elements of tuple_a and tuple_b are displayed on the right side of the screen. The output is as follows: ((1, ' a'), (2, ' b'), (3, ' c'), (4, ' d'), (5, ' e')).* [Video description ends]

Observe the nested tuples within the outer tuple, you can see that every first element of the nested tuple comes from the first tuple that we have zipped up.

[Video description begins] *She highlights the first tuple in the output containing the elements 1 and a. There are two arrows pointing to the digit 1 in tuple_a and the character a in the tuple_b respectively.* [Video description ends]

And the second element comes from the second tuple that we have zipped up. The number 1 from the first tuple is paired with a from the second tuple, 2 from the first tuple is paired with b from the second tuple, and so on. If you want to extract the original tuple from the zipped result, simply call zip once again, pass in the result with an asterisk at the beginning.

[Video description begins] *The 16th code cell displays the following statement: tuple_x, tuple_y = zip(*result).* [Video description ends]

For now you can ignore what this asterisk means, and assign the result to tuple_x and tuple_y. If you print out the values of tuple_x and tuple_y, you'll see these will be the original tuples which you zipped up to get the zipped result.

[Video description begins] *The 17th code cell displays the following statements: print('tuple_x :', tuple_x). Below it is the statement: print('tuple_y :', tuple_y). The output displays, tuple_x :, followed by the elements of the tuple_x. The next line displays, tuple_y :, followed by the elements of the tuple_y.* [Video description ends]

The zip operation is extremely useful to bring two ordered collections together.

[Video description begins] *The 18th code cell displays the following statements: list_a = [6, 7, 8, 9, 10]. Below it is the statement: list_b = ['a' , 'e', 'i', 'o', 'u'].* [Video description ends]

And the zip operation doesn't work just with tuples, can also work with lists. Here we have two lists, lists_a and list_b. A list of numbers and a list of characters. Let's invoke the built-in zipped function on the two lists. Pass in list_a and list_b to zip.

[Video description begins] *The 19th code cell displays the following statement: zipped_list = zip(list_a, list_b). The 20th code cell displays the following statements: result = list(zipped_list). Below it is the statement: print(result).* [Video description ends]

Store the result in zipped_list, and make a list of the zipped_list. And this result is a list of tuples.

[Video description begins] *The output displays the following: [(6, ' a') , (7, ' e'), (8, ' i'), (9, ' o') , (10, ' u')].* [Video description ends]

The result is a list here because we converted the zipped list variable to a list, and notice that every element within the list is a tuple and each of these tuples contain two fields. The first field from the first list, the second field from the second list, list_b. Let's try the zip operation again so that we understand it better. Here we have two tuples a and b, a contains a tuple of names and b contains a tuple of positions within an organization.

[Video description begins] *The 21st code cell displays the following statements: a = ("John", "Charles", "Mike"). Below it is the statement: b = ("Manager", "Supervisor", "Engineer"). At the end is the following statement: x = zip(a, b).* [Video description ends]

Invoke the zip function on a and b and store the result in x. Now this zipped result can be converted to a tuple or a list, it can also be converted to a dictionary.

[Video description begins] *The 22nd code cell displays the following statement: print(dict(x)).* [Video description ends]

Pass in the zipped result x into a dictionary, and the result will be something that makes sense intuitively. The keys of this dictionary are all of the fields from the first tuple, John, Charles, and Mike.

[Video description begins] *The output is as follows: {'John': 'Manager', 'Charles': 'Supervisor', 'Mike': 'Engineer'}. The lists a and b display towards the right side of the screen.* [Video description ends]

And the corresponding values are all of the fields from the second tuple, Manager, Supervisor, and Engine. Let's get some more practice using the zip operation. I'm going to set up two lists and a tuple here a numbers list, a string list and a numbers tuple.

[Video description begins] *The 23rd code cell displays the following statements: numbers_list = [1, 2, 3]. The numbers_list contains numeric values. Below it is the statement: str_list = ['one' , 'two', 'three']. The str_list contains one, two, and three in lowercase characters. The third statement is as follows: numbers_tuple = ('ONE', 'TWO', 'THREE', 'FOUR'). The numbers_tuple contains ONE, TWO, THREE, and FOUR in uppercase characters. At the end is the following statement: result = zip(numbers_list, numbers_tuple).* [Video description ends]

And the result here is going to be zipping up of the numbers list and the numbers tuple.

[Video description begins] *The 24th code cell displays the following statement: print(result).* [Video description ends]

I stored the result, in result. If you see result it's simply a zip object and this zip object can also be converted to a set.

[Video description begins] *The 25th code cell displays the following statements: result_set = set(result). Below it is the statement: print(result_set). The numbers_list, str_list, and numbers_tuple elements are displayed on the right side of the screen.* [Video description ends]

Let's see what the result set looks like.

[Video description begins] *The output displays the following: {(2, 'TWO') , (3, 'THREE'), (1, ' ONE')}. The words TWO, THREE, and ONE within the single quotes are in uppercase characters.* [Video description ends]

Result_set which we got from the zipped object is basically a set of tuples, where every inner tuple gets its first field from the numbers_list and the second field from the numbers_tuple. The numbers_list and the numbers_tuple have been zipped up together. It's also possible to zip together three collections.

[Video description begins] *The 26th code cell displays the following statement: result = zip(numbers_list, str_list, numbers_tuple).* [Video description ends]

Here we've passed in three collections as input arguments to the zip function. The numbers_list, the string list, and the numbers_tuple. The zipped result is stored in the result variable, and let's convert this result variable to a set. Remember you can also convert it to a tuple or a list if you want to.

[Video description begins] *The 27th code cell displays the following statements: result_set = set(result). Below it is the statement: print(result_set). The numbers_list, str_list, and numbers_tuple elements are displayed on the right side of the screen.* [Video description ends]

The result set, as you might expect, is a set of tuples. The first field in every nested tuple comes from the numbers_list, the second field comes from the str_list, and the third field comes from the numbers_tuple.

[Video description begins] *The output displays the following: { (1, ' one', ' ONE'), (2, ' two', ' TWO'), (3, ' three', ' THREE')}. The first field contains digital values: 1, 2, and 3. The second field contains one, two, and three in lowercase characters. The third field contains ONE, TWO, and THREE in uppercase characters.* [Video description ends]

Introducing Other Complex Data Types

[Video description begins] *Topic title: Introducing Other Complex Data Types. Your host for this session is Janani Ravi.* [Video description ends]

Let's move on to some more interesting data types in Python. So far, we've worked with only individual types, one integer, one floating point, a single string. We'll now move on to collections of types. And the first of the collections that we'll study is the list in Python.

[Video description begins] *A jupyter notebook titled ComplexDataTypesInPython displays in the browser window. The first Input code cell displays the following statement: num_list = [1,2.5,'num' ,True]. Below it is the statement: print(num_list).* [Video description ends]

Lists can be defined as an ordered collection of elements where the order matters or the order is significant. Here we've assigned a list to the num_list variable. Notice that a list in Python is created using square brackets, and we have a number of elements separated by commas within those square brackets.

[Video description begins] *She points to the opening and closing square brackets and then points to the commas separating the elements in the list.* [Video description ends]

Observe another interesting thing about Python list. It can contain elements which are of different data types. Here we have a list with an integer that is the number 1, a floating point, the number 2.5, a string called num, and a Boolean value True. When you print a list in Python, it'll print the elements in the same sequence. The sequence does not change unless you explicitly change it in some way.

[Video description begins] *The output displays as follows: [1,2.5,'num' ,True].* [Video description ends]

A list is a special data type in Python. If you pass a num list to the type function, you'll see that the type of this variable is list.

[Video description begins] *The 2nd code cell displays the following statement: type(num_list). The output displays the text: list.* [Video description ends]

Now, individual elements within the list can be accessed by using square brackets.

[Video description begins] *The 3rd code cell displays the following statement: type(num_list [1]).* [Video description ends]

So num_list[1] will give you the element at index position one. Here we want to know the type of the element at index position 1 in num_list. And you'll see that it is of type float.

[Video description begins] *The output displays the text: float.* [Video description ends]

And something should immediately strike you here. The floating point number 2.5 is actually the second element in the list. And this is accessed using the index position 1. This is because, just like with strings, the first position in a list is at index 0. List indices always start at 0 in Python. Just like with strings, the first character is at index positions 0, with list, the first element is at index position 0. So index position 1 refers to the second element which is of type float.

[Video description begins] *She points to element 2.5 in the num_list*. [Video description ends]

If the first element of the list is at index position 0, it stands to reason that the last element in the list is at index position length of the list minus 1.

[Video description begins] *The following text displays at the bottom-left corner of the screen: The last element is always one less than the length of the list*. [Video description ends]

So if a list has four elements, then the last index position is 3. Let's get the type of the element at index position 3, type of num_list [3] will give us Boolean. This is the data type of the last element in the list, the Boolean value True.

[Video description begins] *The 4th code cell displays the following statement: type(num_list [3]). The output displays the text: bool*. [Video description ends]

Let's get introduced to another data type in Python, the tuple.

[Video description begins] *The 5th code cell displays the following statement: num_t = (1,2.5,'num' ,True). Below it is the statement: num_t*. [Video description ends]

Here num_t is assigned a tuple value. A tuple can be differentiated from a list based on the fact that the elements of a tuple are enclosed within brackets, and not square brackets. These are parentheses enclosing the same four elements.

[Video description begins] *She points to the opening and closing curved brackets*. [Video description ends]

Tuples like lists are also used to represent an ordered sequence of elements in Python. The main difference between tuples and lists is the fact that tuples are immutable. Once you've created a tuple, you can't change the individual elements in the tuple. You can't add new elements, you can't remove elements. A tuple once created, remains as is. Let's see what num_t contains. You can see that it's a tuple with the elements enclosed within brackets. And once again, Python tuples can hold elements of different data types.

[Video description begins] *The output displays as follows: (1,2.5,'num' ,True)*. [Video description ends]

In order to differentiate tuples from lists, the elements of a tuple are often referred to as the fields of a tuple. And that's how I'll refer to this moving forward. If you print out the type of the num_t variable, you'll see that it's of type tuple.

[Video description begins] *The 6th code cell displays the following statement: type(num_t). The output displays the text: tuple*. [Video description ends]

You can access the individual elements or fields of a tuple using square brackets exactly like you did with list. Index positions for tuple fields also start at 0.

[Video description begins] *The 7th code cell displays the following statement: type(num_t [2])*. [Video description ends]

So here, when you're trying to access the field index position 2, its type is string, because this is the string num that you have specified in quotes. This is the field at index position 2.

[Video description begins] *The field 'num' is highlighted in the num_t tuple on the code cell 5. There is a blue-colored arrow pointing to index 2 on the code cell 7. The output displays the following text: str*. [Video description ends]

Another data type in Python is the dictionary which is a mapping of key value pairs.

[Video description begins] *The 8th code cell displays the following statement: my_dictionary = {"a": 25, "b": 3.4 , "c": 20}. Below it is the statement: my_dictionary.* [Video description ends]

Dictionaries are specified using curly braces. Dictionaries are typically set to contain records. Here is a dictionary with three separate records, and every record here is a key value pair. The key for the first record is the string a, the corresponding value is 25.

[Video description begins] *She highlights the first key-value pair, containing the string a and the value 25 to indicate a record.* [Video description ends]

Notice that every key value pair is separated using a colon.

[Video description begins] *She highlights the colons for all the key-value pairs.* [Video description ends]

Every record is separated from other records using the comma. And all of the records are enclosed within curly braces.

[Video description begins] *She highlights the commas and then the opening and closing curly brackets.* [Video description ends]

So the second key value pair mapping, the key is b, the value is 3.4. The third key value pair mapping, the key is c and the value is 20. So here is what the dictionary looks like.

[Video description begins] *The output is as follows: {'a': 25, 'b': 3.4 , 'c': 20}.* [Video description ends]

The cool thing about a dictionary, which we'll study in more detail later on, is that you can look up values using their key. If you check the type of the my_dictionary variable, you can see that it has data type dict.

[Video description begins] *The 9th code cell displays the following statement: type(my_dictionary). The output displays the text: dict.* [Video description ends]

The last data type that will be introduced too here is the set.

[Video description begins] *The 10th code cell displays the following statement: a = {5,2,3,'1',4 ,4}. Below it is the statement: a.* [Video description ends]

A set is simply a number of elements specified within curly braces. A set in Python is an unordered collection of unique elements. Observe the set a that we have initialized here. Observe that one of the elements in the set, the element 4 has been repeated twice.

[Video description begins] *She highlights the set a on the code cell 10 and then highlights the last two elements in the set.* [Video description ends]

But when you print out the value of a, you'll see that the element 4 occurs exactly once.

[Video description begins] *The output is as follows: {'1',2,3,4,5}. Element 4 appears only once.* [Video description ends]

You can also see that the elements of the set are not printed out in the same order in which we had specified those elements.

[Video description begins] *She highlights the output and the elements of the set a on the code cell 10.* [Video description ends]

The set is unordered, you cannot rely on the ordering of elements in the set. And if you check the type of the variable a, you'll see that it's of type set.

[Video description begins] *The 11th code cell displays the following statement: type(a). The output displays the text: set*. [Video description ends]

This is once again a special data type available in Python.

Exercise: Lists, Tuples, Similar Yet Different

[Video description begins] *Topic title: Exercise: Lists, Tuples, Similar Yet Different. Your host for this session is Janani Ravi*. [Video description ends]

Let's recall some of what we learned in this course. In this exercise, you will first list some similarities between lists and tuples, and then you'll move on and list some differences between lists and tuples. You'll then write some code to create a Python list and perform access update and slicing operations on that list.

You'll then create a Python tuple and perform the same operations, access and slicing operations on the tuple. At this point in time, I suggest you pause this video and try and answer this questions yourself. Now hopefully you've had some time to think about answers.

[Video description begins] *Solution* [Video description ends]

Let's discuss some of the similarities between list and tuples, your list might be a little different and that's fine. Both lists as well as tuples are ordered collections of elements. It means that the order in which the elements are specified either in a list or in a couple are significant and are retained by the list and the tuple. So if you have a list or a tuple with the numbers one, two and three, this ordering will be preserved. Now, lists as well as tuples can contain elements of different data types in Python.

So you can have a list which has a Boolean element, a floating point, a integer, a string, a character, anything. And the same is true of tuples as well. Elements which are stored in a list or a tuple can be accessed using index values. And index values for both lists as well as tuples start with 0. Elements in a tuple are often referred to as fields. So fields also start at index 0. Many of the operations that you can perform on lists can also be performed on tuples, such as slicing operations.

Slicing operations are exactly the same whether you're working with a list or a Python tuple. Now there are of course, syntactic differences between lists and tuples. Lists are specified in square brackets and tuples in parentheses. But the most significant difference between them is the fact that tuples are immutable. Tuples once created cannot be edited. You can't change the element at any location. You can add new elements, you can't delete existing elements.

So if you want to have an ordered collection like a list but you want to prevent accidentally edits on that collection, you'll use a tuple. Let's move on to our next question and see some operations that we can perform on Python lists. Here is a new list that we have instantiated within square brackets.

[Video description begins] *The screen displays the following statement: new_list = ['Amy' , 'Bob', 'Charlie', 'Daisy']*. [Video description ends]

It's a list of names. List elements start at index value zero.

[Video description begins] *The screen displays the following statement: new_list [0]*. [Video description ends]

So if you want to access the first element in a list, you will access the index position 0 within square brackets, and this will give you the first element here, which is Amy.

[Video description begins] *She refers to the output: 'Amy'*. [Video description ends]

Since index value start at 0, this means that the last element in the list is at index length -1. The length of the list here is 4, at new_list of 3 we get the output Daisy. That´s how we access Daisy within a list.

[Video description begins] *The screen displays the following statement: new_list [3]. Below it is the output: 'Daisy'*. [Video description ends]

Slicing operations allows us to access subsets of lists. So new lists of 0:1 will give us all elements of the list starting at index position 0 up to but not including index position 1. The end index is exclusive.

[Video description begins] *The screen displays the following statement: new_list [0:1]. Below it is the output: 'Amy'*. [Video description ends]

This means that new lists of 0:3 will give us the elements at index positions 0,1 and 2, which is Amy, Bob and Charlie.

[Video description begins] *The screen displays the following statement: new_list [0:3]. The output displays the following three elements within square brackets: Amy, Bob, and Charlie. They are enclosed in single quotes and separated by commas*. [Video description ends]

Extracting subsets of a list in this way are referred to as list slicing operations. So new list of 1:4 will leave out the element at index 0, which is Amy and give us, Bob, Charlie and Daisy.

[Video description begins] *The screen displays the following statement: new_list [1:4]. The output displays the following three elements within square brackets: Bob, Charlie, and Daisy. They are enclosed in single quotes and separated by commas*. [Video description ends]

Now let's move on to answering our next question and take a look at some of the operations that you can perform on tuples. Here is a new tuple that we've instantiated containing the same four names.

[Video description begins] *The screen displays the following statement: new_tuple = ('Amy' , 'Bob', 'Charlie', 'Daisy')*. [Video description ends]

Observe that tuples are instantiated using parentheses, not square brackets. Elements of a tuple are accessed using indexes, just like elements of a list. You'll use square brackets and index position start at zero. So new tuple of 1 will give you Bob.

[Video description begins] *The screen displays the following statement: new_tuple [1]. The output is Bob within single quotes*. [Video description ends]

If you use square brackets and specify an index, and this applies to both lists and tuples, so we are seeing it with tuples here. And there is no element at that index. That is, the index is out of range. You'll get an out of range error from Python.

[Video description begins] *She refers to the statement: new_tuple [4]*. [Video description ends]

Just like with lists, you can also perform slicing operations on tuples to access a subset of elements present in the tuple. New tuple of 0:2 will give you the first two elements at indexes 0 and 1.

[Video description begins] *The screen displays the following statement: new_tuple [0:2]. The output displays the following two elements within curved brackets: Amy and Bob. They are enclosed in single quotes and separated by a comma*. [Video description ends]

The end index here is exclusive you'll now get the element at index 2. Once again, new_tuple of 1:3 will give you the elements at indexes 1 and 2, which is Bob and Charlie.

[Video description begins] *The screen displays the following statement: new_tuple [1:3]. The output displays the following two elements within curved brackets: Bob and Charlie. They are enclosed in single quotes and separated by a comma*. [Video description ends]