Module 2
Video Transcripts

Video 1: Introduction to Python (3:28)

Hey there! Welcome to introduction to Python. In this video, you'll learn what Python is, why should you learn it, and where you can write Python code. Let's get started. First, let's discuss what Python is. Python is a multipurpose programming language. It's very beginner friendly. So, if you don't have prior coding, experience, Python is a great place to start.

The syntax is simple and easy to understand. There are many ways you can use Python. For example, you can use it for data analysis and visualization. When you use Python, you can save your code and reproduce your analysis quickly when you get more data. You can use this to analyze things like products, metrics, or results from a satisfaction survey in just a few seconds. Python also runs quickly, so you can use it with huge data sets and complex calculations.

In other reasons to use Python is to automate routine tasks. If you have an email that you send out at certain points during your workflow, for example, you can write a Python script that allows you to quickly fill out the template rather than rewriting the email each time. Python is also great for building apps; Instagram, Dropbox, and Spotify are just a few examples of apps written with Python. Finally, Python can be used for machine learning.

There are Python libraries that include pre-written functions for machine learning applications. That means you don't need to write your code from scratch each time. There are a few versions of Python, but the most recent is Python 3. So, we'll be using that throughout the course. While there are other programming languages that can do what Python does, Python is very attractive because it's efficient, meaning it can do tasks more quickly and with fewer lines of code than other popular programming languages.

It's also cross-platform, meaning you can use it to build and run apps on Windows, Mac, or Linux. Additionally, Python is open source, that means the source code is free to the public. Why does that matter? Well, that allows you or anyone to use Python for free, which leads to a large collaborative community, that brings me to my next point. Python has a huge community. That means you can easily find answers to your Python questions online or have someone help you if you're stuck.

Finally, there are lots and lots of Python libraries. Libraries are collections of pre-written functions, so you can perform many actions without writing the code yourself. In this course, we'll look at some of the most common libraries like Pandas and NumPy. So, now that we know what Python is and why we should learn it, you may be wondering where you can write Python code.

Python code can be written in a Python shell, which is a program where you can write Python commands and see the outcome. You can test this out at python.org/shell. For example, if I ask Python to 'print('Hello, world!')' it does exactly that. This also works as a nifty calculator. However, if you want to run multiple Python commands, called a Python script, you'll need another tool. A common tool that's used is called a Jupyter Notebook. You'll learn how to use Jupyter Notebooks throughout the course.

****

Video 2: Running Jupyter Notebooks (1:07)

In this video, we'll talk about what Jupyter Notebooks are, and how you can run them on your own computer. So, what is a Jupyter Notebook? A Jupyter Notebook is an open-source web app where you can write code or markdown in cells. You could include equations or images as well. When you write code, you can see the results right in the notebook, so you can iterate on your code right where you're writing it.

The quickest way to get started with Jupyter is to first install Anaconda. Go to anaconda.com/download. Click download, select your operating system, and then download the latest version. Follow the instructions for installation, and then once Anaconda is installed, open up the Anaconda navigator, it will look like this. From there, click Launch under Jupyter Notebooks, and this will open up a Jupyter Notebook in your browser.

****

Video 3: How to Use a Jupyter Notebook (4:48)

In this video, we'll look at how to use Jupyter Notebooks where you can actually write some Python code. Recall the Jupyter Notebooks allow you to write code or mark down. All the things you write in a notebook or written in cells. To select the cell, either click on it or use your up and down arrow keys. There are few types of cells that you'll see in a Jupyter Notebook, but the two most common are markdown cells and code cells. Markdown cells have plain text that can easily be styled.

For example, to make text bold put two asterisks on both ends of it, to italicize use underscores. A bullet point can be made either with an asterisk or a dash, and headers can be created with hash symbols. The more hash symbols, the lower in the hierarchy the header is. There are many ways you can style your Markdown text. The other type of cell you have is the code cell, and just as the name suggests, in a code cell, you can write code.

In Python, you can comment out your code by including a hash symbol before it, any code that's commented out won't run when you run your code cell. This is useful for writing comments describing what your code does so that others can easily use your code, or as you're learning to remind yourself what the code does. You could also use it if you're testing your code and want to see what will happen if you take out a certain line.

So, in this example, the line that says 'print ('Hello, World!')' will run, but a line that has example of a code cell will not. To change a cell from a code cell to a markdown cell, use the keyboard shortcut "M". To change it back to a code cell, use the keyboard shortcut "Y". Next to code cells, you'll see the word 'In'. You won't see that in your markdown cells. All of your cells can also operate in either edit mode or command mode.

To get into edit mode, hit "Enter" on your keyboard. You'll see that the cell turns green. Now, you can write in the cell to go back to command mode, hit "Esc" on your keyboard. In command mode, you can literally command the notebook to do certain things. You can insert cells, delete cells, switch from markdown to code, and so on. So, now that we're in command mode, let's talk about inserting or deleting cells. We'll want to work with keyboard shortcuts to save time. To insert a cell above the cell you selected, hit "A" for above.

To insert a cell below the one you selected, hit "B" for below, and to delete a cell, hit "D" on your keyboard twice. Now, let's talk about actually running code. I'm in this code cell,

and I'm going to hit "Enter" to go into edit mode. Hit "Shift" and "Enter" to run the code. If your code has output, you'll see the output right below your code cell. But what if you have multiple lines of code in the same cell? For example, let's say you're helping your little brother with his math homework. You want to show him '2+3' and '100+102'.

So, you write this all out and you run it. Notice that it will only show the output from your last line of code. So, your brother won't see what '2+3' is from this code. You'll have to run each line separately in a different cell. Another thing we want to talk about is stopping code. In the two examples here, we have a very straightforward code. It doesn't take a lot of time to run, but we get into things like loops, you'll see that you can actually accidentally create an infinite loop that will keep running and running, infinitely.

So, in order to stop the code from running, you can click stop up here or click Kernel and interrupt. Now, to review, we have covered a lot of different keyboard shortcuts. "Y" will change a cell into a code, "M" will make it a markdown cell. "Enter" will get you into edit mode, and "Esc" will get you into command mode.

You can insert cells by hitting "A" or "B" and delete cells by hitting "D" twice. To run the code, you hit "Shift" and "Enter" together. But there are a lot of other keyboard shortcuts you can learn. To see them click Help and Keyboard Shortcuts. This is also helpful if you forget what the keyboard shortcuts are because they're listed right here. And our final keyboard shortcut we'll see is "H". "H" will just pop up the rest of the keyboard shortcuts for you.

****

Video 4: Basic Data Types (4:49)

In this video, we're going to learn about Python data types and why they matter. A data type is a category of data. Any data that you could possibly have, whether it's a number or a word, can be categorized into a data type. Some common data types in Python are Integers, Floats, Strings, and Booleans. An integer is a whole number. A float is a number with digits after the decimal point. A string is text data created using quotation marks, and a boolean is a binary value represented as True or False.

To check the data type in Python, type the function 'type' and in parentheses, enter the data that you're testing. Let's look at some example. Here we see that '1' is an integer, whereas '1.0' is a float. 'Star Wars: The Rise of Skywalker' is a string. Here, it's important to note that this data type is a string because it's surrounded by quotes, not because it's made up of letters. For example, if we create '3' in quotation marks, it's a string, even though the contents of that string is a number.

Finally, 'False' is a boolean as is 'True'. Here, it's important to note that the F in False and the T in True have to be capital for proper Python syntax. So, why does it matter what type your data is? Well, depending on the type of data, there are certain operations you can or can't do on it. And it's trying to do an operation that doesn't work in a certain data type will throw an error. Let's see this in action. Here we have two booleans, False and True.

What happens when we add them together? It works. False is evaluated to 0, and True is evaluated to 1. So, when we add them, we get one. The data type of that output is an integer. This works exactly the same as adding integers, 1 plus 2 equals 3. However, what happens when we try to add strings? They have no numerical value. So, the plus operator

works differently here. Instead of doing math, plus operator concatenates the two strings and outputs a new strain that's a combination of the two you added.

So, here we learn that operators work differently on different data types. In fact, some operators only work on certain data types. If you look at the minus operator, for example, we can use it with booleans, integers, and floats as expected, but when we try to use it on strings, we get a TypeError. The TypeError message makes it pretty clear that we can't use a minus operator on two strings. So, what happens if you want to change the type of our data? Python has built-in functions for that.

Let's take a look at the number 15. '15' is an integer. If we want to convert it to a float, simply write float and put 15 in parentheses. However, if we check the type now, we see that it's still an integer. What happened? Python doesn't automatically store changes this way. If you want the change to be updated, you first need to assign a value to the variable. Let's see an example. Here, we say 'x = 15'. Checking the type, we see that 'x' is an integer as expected.

Now, if you want to store 15 as a float rather than an integer, we need a new variable. Let's call it y. If we say that y is 15 as a float, the type of x doesn't change, but we have a new variable 'y' that's a float. We can do this with any data type. For example, to change the data to a string, we type 'str' and put the data following it in parentheses. To change to an integer use 'int' and to change to boolean use 'bool'.

Let's spend some time now looking at booleans because the conversion of data to a boolean isn't totally intuitive. Let's say we have the integer '123'. If we change it to a boolean, we get 'True'. Why is that? Because it's not zero. Zero will always evaluate to False, whereas any other number will evaluate to True. Even negative numbers will evaluate to True. Okay, looked at four data types, integers, floats, strings, and booleans. But there are many, many others. Other key data types includes Lists, Tuples, Dictionaries, and DataFrames.

****

Video 5: Comparison and Logical Operators (4:43)

In this video, we're going to take a closer look at the boolean data type. You may remember that booleans are either True or False, and that True evaluates to one, and False evaluates to zero. So, whenever you do any kind of math with booleans, it's as if you're doing math with one and zero. Converting an integer or float to a boolean will create a False if your integer or float is zero, and a True if your integer or float is any other number. Let's take a look at a few more examples.

Let's assign x to True and y to False. What do you think will happen if we do 'x+2'? Since, x is True, which evaluates to one, we'll get '3'. If we do 'y+2', we'll get '2'. And if we 'y+x', we'll get '1'. Pretty simple, right? Now, as a final review of booleans before jumping into new concepts, let's create a new variable 'm' and assign it to be the int(x). What do you think m will equal to? Again, since x is True, the integer of x will equal 1, similarly, if we create a variable 'n' and assign it to be the int(y), it'll equal 0. Since y is False.

Now, let's look at some ways booleans can be used. When you want to compare two or more values, your output will be a boolean. To check if two values are equal; for example, we use two equal signs. Here, we see that 4 does not equal 5. That's pretty obvious, but it'll

come in handy when you're learning to clean and analyze data. You might want to remove redundant data, for example.

Other operators you'll need to know are '< Less than', '> Greater than', '>= Greater than or equal to', '<= Less than or equal to' and '!= Does not equal', which is denoted as an exclamation point into equal sign. Okay, let's jump into the last segment of this video, logical operators. Similar to comparison operators we just saw. Using logical operators will output a boolean value. Two common logical operators are 'and' and 'or'. Let's start with the 'and' operator. This will evaluate to True if everything in your expression is True. This will make more sense with some examples.

If we type 'True and True', our output will be 'True'. However, if we type 'False and False', our output will be 'False'. Similarly, 'True and False' evaluates to 'False'. Again, using 'and' you will only get True if everything in the expression is True. Now, let's look at 'or'. This will evaluate to True if at least one of the values is True. So, 'True or True', will be 'True'. 'True or False', will also be 'True'. Let's look at some extended examples.

Here we have an expression using only 'ands'. The values we have here are True, True, False, and True. What will this evaluate to? Well, with 'ands' all values have to be True for the output to be True. Since we have one False here, this will evaluate to False. And now, just for fun, let's try a combo of 'and' and 'or'. What do you think will happen here? Let's go in order. 'True and True' is 'True'; hence 'True or False' is also 'True'. So, this will evaluate to True.

Another thing to note is that logical operators can be used along with comparison operators. For example, if we have 'x = 5', and 'y = 10', we can test if 'x>3' and 'y<20'. Since both of these expressions are True, our result will also be 'True'. Now, that you're probably comfortable with 'and' and 'or', let's look at one final logical operator, 'not', 'not' simply reverses the boolean. So, 'not True' is 'False', and 'not False' is 'True'. Take some time to play around with combining, 'not' with 'and' and 'or'. You can also try combining these with different comparison operators.

****

Video 6: Lists and Indexing (5:28)

In this video, we'll look at a new data type, a list. We'll discuss what a list is and learn how it can be used to store and retrieve data. A list is a data type that is a combination of elements. It's denoted with square brackets containing the elements of the list. All of these elements are ordered. Let's look at an example. Here, I created a sample list called 'my_list'. It contains the elements '1, 2, 3, 4' and '5'. Because I use square brackets when creating the variable, 'my_list', the data type of 'my_list' is as expected a list.

Lists can be used to store ordered data. For example, if I recorded the temperatures every day for six days, I can store it in a temp, in a list called 'temperatures'. The great thing about lists is they can store any kind of data. In this example, I created a list of integers for the temperatures, but that makes it hard to tell which date each temperature is associated with. A better way to create the list is to include the date as a string right before each temperature.

Here, I created a list of strings and the integers to store the date and its temperature for six days. I can combine this with any other data type. Let's say, I was curious to see how my

mood was associated with the temperature. Each day, in addition to recording the temperature, I also recorded whether or not I was in a good mood using a boolean. 'True' means I was in a good mood, and 'False' means I wasn't.

I'm storing this in a list called 'dec_temperatures_mood'. I mentioned previously that lists are ordered. That means that each element in the list is in a specific position identified with an index. In Python, indexing starts at 0. So here, ' "Dec 10" ' is at position 0, '20' is in position 1, 'True' is in position 2, and so on. This is useful because it makes it easy to retrieve data from the list. If I want to know what's at a certain position, I type the name of the list, followed by square brackets, and the index I'm interested in.

Here, I'm checking what's in position 0, and I see that it's 'Dec 10'. Here, I'm checking what's in position 5, and I see that it's 'True'. This is particularly useful for very long lists, where you can't easily glance at your data, and see where everything is. You can also combine indexing with a type function to see what type of data you have at a certain index. Here, I see that an index three, I have a string value. That makes sense because, at 0, 1, 2, 3, I have the value ' "Dec 11" '.

Okay, let's shift gears a little while our lists can be easily indexed, you may notice that it's not very organized. It's hard to see which date goes with which temperature and mood. Imagine how hard this would be with an even longer list. One way we can clean this up is to create a list of lists. Remember, that a list is a data type that can contain elements of any data type. That means a list can contain other lists. Rather than listing all the elements we did, let's create mini lists for each data.

I'm going to copy and paste the original list here, and add square brackets around each set of elements that I want in each mini list or sub-list. If we print the list now, we see that it's much easier to read. If I index the list now, I see that I get a sub-list at the position I selected. For example, the index one will give me the values for the second date in my data.

Remember, this is because indexing in Python starts at 0, not 1. To review, in this lesson, we learned what a list is, and saw some examples. We talked about how to index the list using square brackets, and how we can keep our lists more organized using sub-lists.

****

Video 7: Advanced Indexing (3:10)

In this video, you'll learn how to subset several data points at a time called slicing. This is helpful when you want to look at only a part of your data to answer a specific question or to select a smaller sample before analyzing the entire data set; I'm including our list of lists here. As a reminder, each sublist includes a date, the temperature on that date, and a Boolean representing whether or not I was in a positive mood on that date.

There are six sublists in the list, one for each day that I collect the data. Each of these sublists is an element in the list. To see more than one elements in the list, I start out the same way I do for regular indexing. I write the name of the list I wish to index followed by square brackets. Now, instead of entering one index, I'll enter two indexes separated by colon. The index before the colon is where I want to start slicing, and the index after the colon is where I want to end slicing.

Let's try '0 : 1'. What will that give us? You might expect that will give us the element index zero and one, but that's not right. Here, we only see the data at index 0. That's because the index before the colon or the start bound is inclusive, but the index after the colon or the

stop bound is exclusive. So, what we're really telling Python to do here is give us the data from index 0 to 1, not including 1.

If you want to see the data for the first two dates, we'll have to slice from 0 to 2, like this. If you want to see the last elements in the list, you can use the index negative one. You can also slice and reverse, meaning from the end of the list rather than from the beginning with negative indexes. For example, the slice from '[-3 : -1]' will give you the elements in the third to last position to the second to last position.

Remember, you won't get the last element, this way because the stop bound is exclusive. So, how do you slice elements to include the last one? You can subset data with no stop bounds. For example, if you slice from index '[1 :]' you will get all the elements in the list from index one until the end of the list.

If you slice with no stop bound, you will get all the elements in the list before that index. Here, we'll get all the elements before and not including index three. To review, we can slice our data using square brackets, separating the start bound and the stop bound with a colon. We can subset and reverse using negative numbers, and we can subset without a start or stop bound to get all the elements after or before a certain index.

****

Video 8: Updating Data in a List (4:19)

In this video, you'll learn how to add, delete, and update data stored in a list. Let's continue using our list of lists that includes dates, temperatures, and whether I was in a positive mood. After I made the list, I collected more data on 'Dec 16'. On that date, it was '18' degrees, and I was not in a positive mood. I'd like to add this information to our list. To do that, I'll write the name of our list 'dec_temperatures_mood' followed by '+ [ ]', this indicates I'm going to add whatever is in the brackets into the list.

Since I'd like to add a list with the information I just mentioned, that's exactly what I'll put in. Notice that because I added a list to the list, I have to have double square brackets. If I'd forgotten to do that, each element in the brackets would have been added individually, and our list would have looked like this. However, the data we added won't get stored unless they're explicitly assigned. So, even if I print this list out now, I'll see that it looks just like it did in the beginning, no Dec 16th data here.

To save the list of the updated data, assign the changes to the list using an equal sign like this. Printing the list now, we'll see that it was updated. Now on second thought, I actually think I was in a positive mood on Dec 16th. I want to delete that data I just added, and replace it with the correct information. To delete data from a list, we need to use the 'del' function type 'del' followed by '( )'.

In the parentheses, enter the name of the list 'dec_temperatures_mood' followed by '[ ]'. In the brackets, enter the index of the element you want to delete. Since it's the last element, I can enter the index '[-1]'. Now, I'll add the new information. On Dec 16th, it was 18 degrees, and I was in a positive mood. Let's print out our list to check that it updated. That worked, but it wasn't the best way to do that. The 'del' function is great if you just want to remove

an element from the list, but if you want to replace the elements with another one, it's more efficient to use sub setting. Write

out the name of the list, followed by square brackets. In the brackets, enter the index of the element you're trying to replace.

In our case, it's negative one. Set that equal to another set of square brackets and enter the updated information. This way, you can update your list with one line of code instead of two. In the long run, this could save you a lot of time. That's all for this video. You should now know how to add, delete, and replace data in a list.

****

Video 9: Introduction to Tuples (2:04)

Welcome back. We'll now learn about another data type, a tuple. The easiest way to understand what a tuple is, is to compare it to a list. Just like a list, a tuple is a collection of elements. It can include different data types and is ordered so that it can be indexed. However, we also learned that we can add, delete, or replace the elements of a list. That means lists are mutable.

Tuples, however, are immutable, that means they cannot be modified after they're created. So, if you have data that needs to be updated, often use a list. For example, if we are recording and adding the temperature each day, it makes sense to use a list. However, if you have data that's fixed, you'll want to use a tuple. This protects your data since it can't be changed, so you won't introduce any new errors.

Python syntax differentiates between lists and tuples, whereas lists are created using square brackets, tuples are created using parentheses. Let's create our first tuple. Here's an example of a tuple containing the names of the characters from the movie ET. Once we create the tuple, we can't add, delete, or replace anything. However, tuples can be unpacked. Unpacking a tuple means assigning each elements of the tuple to a variable.

To unpack a tuple, write out the names of the variables you want separated by commas, and assign them to the tuple. Here, we create variables with the names of actors who played each character in ET. Let's see what happens when I print out these variables. If I 'print(drew)', I get 'gertie'. If I 'print(robert)', I got 'michael'. This is a really straightforward way of getting the data you need.

****

Video 10: Introduction to Dictionaries in Python (4:03)

In this video, you'll be introduced to a new Python data type, the dictionary. A dictionary is a Python data type that maps a set of objects to another set of objects using key value pairs. Let's compare dictionaries to another data type lists. Let's look at a new example, where I've recorded the dates and the temperature on those dates. From these two lists if you want to get the temperature on December 14th, we have to figure out the index of December 14th in the dates lists, and then find the temperature at the same index in a temperatures list.

To do this, we can use the index method on the dates list. We see that December 14th has an index of four. Now, we'll find the temperature with index four in the temperatures list.

We see it was 13 degrees that day. So, you might notice a couple of problems with this process. First of all, it takes way too many steps to get a simple information we want. Second of all, this makes maintaining our data very difficult. Adding a date to the dates list

in the wrong spot messes up the mapping of all the dates to their temperatures. This could be a nightmare with large data sets.

There are two solutions you should already be familiar with. One of the solutions is to create a combined list with the dates and the temperatures. However, the order of the dates and the temperatures need to be very specific, one right near the other, and any small data entry mistake will mess up the entire list. Another solution you should be familiar with is creating a list of lists.

This is much cleaner, but again, you're losing the independence of each date and temperature, and this would make the data difficult to analyze since each sublist is now one object. Dictionaries solve these problems. In a dictionary, each key is by definition, associated with each value. So, we can declare there are certain date is specific to a certain temperature. Dictionaries are also easy to read, retrieve data from and analyze. Let's look at how to create one. Dictionaries use curly brackets.

Let's create a dictionary called 'dec_temps'. In the curly brackets, we specify each key, and the value separated by colon. The keys here are the dates, and the values are the temperatures, we separate each pair with a column. Now, to retrieve data, all we have to do is write the name of the dictionary, and then square brackets the name of the key.

The output will be the value associated with that key. So, to get the temperature at December 24th, we type 'dec_temps[ ]' and in the square brackets, the string 'Dec 14'. We see that the temperature is '13' degrees. That saved us a lot of time and steps, and also makes our data much easier to maintain.

****

----------**END**----------