# Module 2: Video Transcripts

## Video 1 – If Statements and White Space

Welcome back to module two, and this week we're going to be covering a few different new Python concepts, such as if statements, lists, and loops. We'll start out by talking about if statements. Now, so far, everything we've written in Python has run all the time, but what if we want to create branching path like to check if something is true before doing some other thing, and that's where an if statement comes in. So, we'll start out by creating a new Python file, opening up Atom, and we can go ahead and save this as say, 'if.py' is a pretty good name. We don't need the sidebar, and I'll just dive right into it.

So, we're going to write like a joke-telling script. So, we'll first take, create a new variable called 'answer ='; we'll take someone's 'input' and say '('Do you want to hear a joke?')', okay? Now, depending on what they say, we'll either want to tell them or joke or not, and the easiest way to do this is by using 'if'. So, here's what that looks like, I'll write 'if answer', this variable we created before '=='. So. this is actually '==', and we'll talk about what that does in a second. 'Yes', Now, we'll put a ':' here at the end. So, this pattern or this construction is going to be fairly common. 'if' something and then ':', and that's what tells Python what to do if that's true. Now, if you add the ':' when you hit Enter, you should see that it's automatically Tabbed in.

So, this is not lined along the left side anymore; there's a Tab. And here, I can now do something like write 'print('I'm against picketing, but I don't know how to show it.')', and then, we'll put a little bit of a '# Mitch Hedberg (RIP)' with attribution. So, it's one of his jokes, right? So, this is what the full file looks like; you can pause it and make sure you copy it along properly. Once you've done this, let's go ahead and run this in the command line. Move that over, open up the command line, move that to the left. Now, 'cd' into our code folder, remember how to do that? first 'cd' into 'Desktop', and then 'cd' into the 'code' folder, and now we can run this file, 'python if.py'.

So, as expected, it's asking, 'Do you want to hear a joke?', and if I type, let's say, 'Yes' then it prints out 'I'm against picketing, but I don't know how to show it'. However, if I rerun this file, and I type 'No' then it doesn't print it out, or if I type literally anything else like if I mash on the keyboard, hit Enter, I get nothing. So, how does this actually work? What is this doing? Well, first I want to talk about this '==' here, what does this do? Well, what this symbol does in Python is, it checks to see whether two things are equal. So, it looks at the thing on the left and the thing on the right, and if they're equal, it says true, but if they're not equal, you get back false.

So, to actually demonstrate this a little further, I want to open up the Python interactive mode here in the command line by just typing 'python' and hitting Enter. Because here now, we can play around with, you know, this idea. So, actually, if I do 'answer == 'Yes'', the same way I have here on the right and hit enter, what's going to happen? Well, actually, I get this 'NameError: 'answer' is not defined', and that's because I'm trying to use a variable I haven't created, which I can't do. So alright, first let's do 'answer = 'Yes''. Now, note how similar

these two things look, right? But the first one was checking if they were the same, the second one is creating a variable, variable assignment, which we covered last week.

So, just notice that one little '=' actually makes quite a bit of a difference. So, now 'answer' does equal 'Yes', and so if I type, 'answer == ‘Yes’’, what I get back is 'True'. So, this isn't actually a string, this is true or false, it's something called a boolean in Python, and it's named after the guy, who sort of initially came up with the concept. But boolean, is another kind of thing, we've got numbers, we've got strings, and we have booleans in Python. So, something can be true or false, just like this. Now, let's override the value of 'answer' and say, 'answer = "No"', and there's nothing in Python that says, you can't rewrite a variable, so we just did that. And now, if we do, let's check again is 'answer ==', is it the same as 'Yes' or does it also contain 'Yes', well now I get back 'False', so that's the other kind of thing.

So, how would I want to, if I wanted to check if something is not equal to something else, right? How would I do that? Well, I can do something very similar instead of '==' I do an '!' and then an '=', So, this is the same as check if answer is not equal to. In this case, let's try 'Blue', right? Because answer is 'No', it's going to be 'True' that it's not equal to 'Blue'. However, if I check is it not equal to 'No', I get 'False' because it is equal to 'No'. So, that's how this '==' part works. Now, how does the 'if' part work? Well, the 'if' really just checks to see if something is true or false. So, whatever you put after the 'if', it's going to check, is it true? If it's true, it'll run whatever code is after this ':', and is Tabbed in.

The Tabbing here is actually really important because this is how it knows what part of the following code to run if this part is true. Well, as soon as I unTab stuff like if I come down here to line six, and let's say I do print or do some rest of Python stuff, it'll start always running again. So, it's only the stuff that's Tabbed in immediately after this 'if' that'll only get run if the 'if' part is true. Whitespace, Tabs, and spaces and stuff like that do matter a lot in certain areas in Python, and 'if' is one of those areas. Now, this Tab, just so you are aware. In Python, a Tab can either just be the actual Tab key, where you can use four spaces in place of a Tab.

Sometimes, text editors just convert one into the other and vice versa, and it's not that big of a deal, whichever one you use, except that you can't do both of them in the same file. If you try to use four spaces and Tabs, you'll end up getting an error when you run the code. So, just something to be aware of. Sometimes you copy someone else's code, and they do it one way, but you do it another way, and you'll get that error that pops-up. So, that's it for the introduction to if statements, and let's talk more about what else you can do in an if statement, and before we get out of here, let's just exit out of this Python interactive shell, we can clear this out as well, before we continue.

****

## Video 2 – Else and Elif Statements

So, we can expand on this if statement, right? What if we wanted to do something if ‘answer' is not equal to 'Yes'. So, we have this one check here. Well part of an if statement is the option to add an else statement, which we do like this, so we just write 'else:', and it comes after the 'if' part, but it has to be untabbed. So, if you're still tabbed in, just hit Delete and that should untab you, it has to be at the same line as that initial 'if' and then a ':', and when I

do this, and I hit Enter, it will be tabbed again. And this is where I can write something like, let's say, 'print("Fine.")', okay. So, you can sort of guess how this might work in this case. If I run this code and I say, 'Yes', I do want to hear a joke, then I get the joke, but if I rerun this code, and I write 'No', then I get 'Fine.'

So, I've created this branching path. And in this case, I could write literally anything else except for 'Yes', and what I'll get is 'Fine'. And yeah, just as an interesting note, this is case sensitive, in case you've been wondering, do I have to type it exactly? Well, if I type 'yes' with a lowercase y, I also get 'Fine', that's what 'else' does is any other scenario except for this, triggers the second part of this if, else statement. So, what if we want to add other 'if's' that are more specific. So, let's say, we have one to take care of the case, where the answer is 'Yes', or the answer is 'No'. And then otherwise do some third thing.

What I can do is, I can put a second 'if' in here. So, again after 'if', but before 'else' or in-between this, sandwiched in there, I can write 'elif', and now this is basically the option for me to put in a second 'if' condition. Whereas, 'else' didn't need anything, it just handled all the other cases, elif requires you to put in another thing that's either going to be a True or False. So, I'll say, 'elif answer == 'No':', let's say, 'print('Fine.')', and then otherwise 'print('I don't understand.')' So, this is, we've changed it now so that this last condition is kind of more of a catch-all. And maybe I could tell someone; I don't understand, try again or have it run again or something like that, just an idea. And so, now if I run this code, if I write 'Yes', I got the joke, if I write 'No', I get 'Fine', and if I write say, 'Blue', it'll tell me, it doesn't understand.

The idea of 'elif' is that you can add as many other 'if' conditions as you want. So, I'm creating now a branching set of paths, think of it as maybe doors, differently numbered doors, and you can only walk through one of the doors. The ordering does matter here though, you always have to start with an 'if', and as we saw before, you don't need the 'else' or the 'elif', you can just have an 'if', but if you want a path, where you can only go down one of the options, the next one would either have to be 'else' or 'elif', and I can put more elifs there, but I can only have one 'else' at the bottom, all the way at the bottom.

Now, this is important because there are sort of gotchas that you should keep in mind here, and one of those will be, for example, if I forgot that this should be 'elif'. So, now I have two if statements, one right after the other, so 'if' and then another 'if', and then an 'else', and this is going to behave differently if that was an 'elif'. Can you think about this, and see how this might do something that we don't expect it to? Well, one problem here is if I run this code now, and I say, 'Do you want to hear a joke?

Yes', I'm going to get both the joke because it triggered this part of the if statement to be run, but I'm also going to get, 'I don't understand' because essentially, now I've created two separate sets of if statements, and I'm putting in space here, just to make it clear to you how this might be working now, it's that this is kind of checking on its own. But once this whole check has finished, it's going down here and doing a second check, and say, well 'if answer' equals 'No', then 'print('Fine.')', otherwise in all other cases including 'if answer' is 'Yes', which we checked above, 'print('I don't understand.')' So, that's why it was important that if you do a second 'if', you have to put 'elif' in the beginning, which is what tells Python to sort of chain these together so that only one of those things can happen.

Another thing that you can be aware of, it adds more levels of complexity into this is you could theoretically put 'ifs' inside of other 'ifs'. So, I might have another 'if' somewhere inside of here, and what happens then is it gets more and more tabbed in, or more and more as it's called nested in Python. So, you have nested 'ifs', one nested inside the other, nested inside the other. That can get quite complicated, and we're going to stay away from doing too many nests. But just so you're aware, it is possible to do, and we will start to see things nested within themselves, over the course of this class.

****

### Video 3 – Logic in Python

This is a great time to talk about logic in Python. Well, what do I mean by logic? I'm basically talking about True or False statements or things that can be evaluated or converted into True or False. Things like we saw before; is answer 'yes' or is answer 'no', or is it something else. So, there are all sorts of different truth terms, as we'll call them in Python. These are different symbols that will produce a True or a False. So, two that we've already seen are equal to or not equal to. We've also already seen greater than, greater than or equal to, less than, less than or equal to. So, if you're checking, are two numbers greater or less than each other, you'll get back True or False.

Three additional terms are not, and, and or, and in the next few slides, I'm going to walk through how these might work. So, this is one example. And now, I'm showing you on the slide, if you open Python mode and set 'answer = ‘yes’, then you can do 'answer == ‘yes’’, and that would be True, but 'answer == ‘no’’, will be False. Now, to be honest, I set it philosophy, we had to memorize a lot of truth tables in logic class, and I don't really enjoy them, there's a lot of memorization. I will introduce you to a few truth tables so that you see how they work, but I don't think it's all that important that you memorize them.

Just so you know that they're available here to use as a resource in case you do end up getting a little confused, as logic is something you might have to work with in Python, and it can be a little bit tricky. So, here's a truth table that I've created on the right-hand side, and the idea of a truth table is that in the first column, I have a Python code, and this code specifically relates to the term ‘not’. And then, on the right side, I have the result of running that Python code. So, in the first row, the code is 'not True'. Now, the capitalization here is important.

The not is lower case, but the True is upper case because True and False are actually specific things in Python, and they have capital letters in front of them if you want them to work properly. And the result of running 'not True' is 'False' because it just flips whatever it is, and the result of running 'not False' is 'True', which is pretty intuitive. Here's another one. So the “=” symbol, two equal signs, in the first example, the code is '1 == 1', which is a pretty stupid example, but the result will be 'True' because they're the same. And now it can go is '1 == 0', 'False'; is '0 == 1', 'False'; but is '0 == 0', 'True'.

So, you can kind of see the character of how this symbol might work with all four of those examples. Here's another one, and there's only one more after this one, I promise. So, in terms of 'and', now 'and' will require either True or False on either side, which is different from the last one we saw, where you can put two strings or two numbers, or anything. In this case, you either need True or False or a statement that produces True or False. So, if you

have two things on both sides of 'and' that are both True, the result is True, whereas, in any other case, the result of this 'and', and something on both sides, will be False. Now, an easy way to think about this is to replace the words True and False with like an actual statement that is obviously true or is obviously false.

So, two examples I think about is the sky is blue, grass is green. Those are both obviously true, and so if I said the sky is blue and grass is green, that entire statement is true. But if I said the sky is blue and grass is red, well, that would be false, even though the first part is true, the whole thing as a whole is false, and that's an example of the second one, where you have True and False, and so on. Now, 'or' is the last one that I'm going to show you as a truth table because 'or' is kind of the opposite of 'and', in that as long as either side is True, the result is True as well.

The only case where using 'or' will give you False as the result is, if the thing on the left and the thing on the right are both False, then the ending, the outcome, ends up being False. So just so you have a little bit of an understanding, if you didn't ever go through truth tables, that's how these different ones will work. And I also want to mention that there's another one called 'in' that you can use to check to see if something is in a list of things. Now, we haven't talked about lists yet, but we'll do that pretty soon, and it is possible to check is something inside of this list, and you'll end up getting True or False. There are all sorts of other ways of getting True and False as well, but that kind of about rounds out the ones that I think are useful for us to know at this point.

****

### Video 4 – Challenge: Logic in Python: Boolean Practice

Now, what I've done is I've actually created a logic practice Python file that you can use to test out your understanding of what we've just covered, and you can go ahead and download it. So, it looks like logic_practice.py. I'll go and move it into my code folder right here, and I'll show you how this works. Another way I can open this up and add them is just drag it onto the symbol. So, this is what logic_practice.py looks like. It has 20 lines of Python code, starting with just 'True and True' and then 'False and True'. And the idea is, it gets increasingly complex in terms of using and's, or's, not, and things like this.

Now, each of these lines, if you run it in Python, will give you back either true or false. So, it becomes one or the other, and it's your job to figure out which one it is. So, I'll walk through the first example with you together. So, 'True and True', if you remember from the truth table that, that should be True, the entire thing. If you have two sides of the and, and they're both True, then the result of that is True. So, these are a few ways you can go through and work on this. One thing I'd recommend is, as you go through, write a comment along the right-hand side with what you think it is, and that way you have all 20 of them, and you can either print that specific line, and what that will end up doing is just print out the actual value of whether it's True or False.

It'll run the whole thing and just print out the result, or you can go into the command line, open up the Python interactive mode, and just paste that entire line in there, and you'll end up getting back the result. So, two ways of doing this. And I recommend going through this file now and see if you can figure it out. And if you're having trouble, I recommend working

your way from the inside to the outside. So, as it gets more complicated, like what we have below, start with as far in as you can get. So, something like this, and then go through and actually figure out you can even replace it in your mind, this '==' that is going to be True, and you can go through it and figure it out that way. So, just to get a little bit of practice, give this one a shot on your own.

****

### Video 5 – Or

So one thing we've noted already is that our file, the answer is case sensitive. So even if I wanted to put in ‘yes’ with a lower case y, it would register 'I don't understand', I have to type it in exactly as it's checked here. Now, there's a few ways of dealing with this, and I'll walk you through in sort of a progression of how I might get to the best solution for this. One way you might think is, well, we've heard about this thing called 'or', so I can check. Is answer equal to capital 'Yes' or lower case 'yes'. So, let's go ahead and add that in. On the right here, we can do 'if answer == ‘Yes’ or:', now here's one question, and this is one place that it's very easy to introduce a bug. If you just put 'or "yes":' lower case, it's not going to work as how you expected. And here, I'll tell you; I'll show you why.

So here, okay, if we just were checking the positive case, meaning does this thing do what we want when we type 'Yes', capital Y or lower case y, oh! It looks like it's working just fine, right? But let's check one of the negative cases, what if we type 'No'? Will I still get the joke? And in fact, if we type ‘Blue’, I'm still seeing the joke. So, I've definitely introduced a bug into my code, and it was a tricky bug because I didn't realize until I checked these other options. So, what's going on? Essentially, it looks like this part of this if statement is always being triggered, so why?

Well, remember how I explained how 'or' works, is it checks to see the value of the things on both sides, 'or' will it return True if either side is true? So, think of how we might be, as, you know, humans, parsing this statement that you see. We're saying 'if answer' is equal to 'Yes', upper case, or 'yes' lower case, but that's not how the computer is parsing this. The computer is separating both parts of the 'or', and it's using that as the separator. So, what it's doing, and I'll put in parentheses here, just to sort of make it clear to you, is it's first evaluating is this true, and in the case of, you know, any of these options besides 'Yes', well, it's False.

However, what's the value of just 'Yes', just the string? It turns out that in Python, any string is True by default. So, if you just put the string value after an 'or' or after an 'and' or before, it's always going to be converted into a True. And if you have an or, whenever either side is true, the whole thing is true, which means that you know, by introducing it this way, I've made this part always run. So, what you really need to do here is, put the entire thing also on the right side of the 'or' part, so you're checking again 'or answer == ‘yes’:'.

There's not an easy way of doing it the other way that we wanted. So, it sorts of look a little long and redundant, and I'll talk about ways of shortening this in the next video. But if I wanted to down here, I could also say 'if answer == ‘No’ or answer == ‘no’:' lower case n, and save. Now, this should do what we expect. So, I could put 'yes', lower case, 'no' with lower case, and anything else just goes to 'I don't understand'. But there's already ways of improving this that you might have thought of, and that's what we'll talk about in the next video.

****

### Video 6 – In

Another way of doing this is to use the 'in' that I mentioned before. So, rather than checking if ‘answer == ‘Yes’ or ‘yes’:’ what I can do is I can check if answer is in and then put a list of both options, ‘[‘Yes’ , ‘yes’]:’ Now, we haven't covered lists yet, and we will shortly, but this is one way of checking. If I wanted to have multiple things I'm checking against, this would probably be the way to go because, you know, you can imagine if I was using or, that would start to get quite long if I was adding more options in there.

Now, what if I wanted yes or any other kind of positive, I can create a list in this case. And I could even, theoretically, take this list out and assign it to a variable somewhere so that I could just add to that variable later on. And we'll see that this will now work. Oops! Now, I entered it into Python mode. Let's exit. python if.py will work if I use lower case 'yes', capital 'Yes', lower case 'no', and so on. But what you usually see in cases like this, especially when you want to make something case insensitive, is you will see a string function used, and I'll show you how that would work.

If answer is in ‘yes’, or capital 'Y' or lower case 'y', an easier way to do this is to convert answer into lowercase first before checking. So, you just say if ‘answer.lower( ) == ‘yes’:’ and if ‘answer.lower( ) == ‘no’:’ This is probably the easiest way to do case insensitivity, and you'll see here now this works with 'yes'. It also works if you do case 'YES' in all caps. So, there's a, there's a few examples of how this is used all the time, and we don't even realize it. And one of them is when you sign up for a website, and you give them your e-mail address as like your username or even a regular username; most of the time, that e-mail address is converted into all lowercase before it's saved into the database.

And that way, you don't have the situation where two different people potentially sign up with the same e-mail address, but one of them typed in as all caps, and the other one didn't. It would be really bad if the same e-mail address was able to create two separate accounts because one was capitalized and the other one wasn't. So, typically, when you put into a database, it's all lowercased before it's done that. Now, it could have been all capitalized, and you can just always capitalize things before saving it in, but this is just the standard that most websites and most different services have ended up resulting to. So, doing this string conversion before checking is a very common practice.

****

## Video 7 – Lists

Finally, we've come to lists. So, I want to introduce how those work in Python. I'll create a new file here just like we have been before and let's call this one lists.py. So, leave a little bit of a comment and say, '# In Python, lists are ways of grouping together', I'll put a new line, so it's not too long, '# similar things (usually)', and I'll give you a few examples of different lists. Let's create a new list, and we'll call it the count as the variable. And now, 'the_count = [1, 2, 3, 4, 5]'. So, just follow along with how I'm doing this, and then I'll explain the different ways that a list might be created or the different intricacies here.

Let's create a second list, and we'll this one 'stocks'. So, stocks is ('FB', 'APPL', 'NFLX', 'GOOG'). Alright, the famous FANG stocks. One final list, let's create, and we are going to call this one 'random_things'. So, one thing you'll notice so far is by creating a list, when you do that, you always do it by square brackets on the left side and all the way on the right side. So, a list always has square brackets around it, and then you put these things inside of it followed by commas, and these things are called 'elements', typically. So, every list has multiple elements.

Now, what kind of a thing can be inside of a list? Well, it turns out that lists can accept any number of different things and in fact, you can have different kinds of things within one list. So, in this last case, I'll say, 'random_things' has '55', it's got '1/2', which will actually end up being converted into '0.5', a float.

Then we've got in other word the "Puppies", and then finally, I'm actually going to put the 'stocks' variable as the last element of this list, and that will end up being replaced with the entire list that we had before. So, we're actually going to be putting a list inside of a list. So yeah, things can get, you know, kind of meta in that sense. You get lists with lists inside of them, and all the way down. So yeah, let's talk about what you can do with lists in the next few videos.

****

## Video 8 – Building Lists from the Ground Up

So, I've just shown you how you can create lists with things already inside of them. And in this case, we've already started out with these things in there, but sometimes it's actually helpful to create an empty list and to add things into it or take them out depending on what your code is doing. So, we can start out with an empty list by just, let's create a variable called 'people = []' nothing, alright? Now, I have an empty list, but the cool thing about an empty list is that there are ways of adding things into it now that I have this variable here.

So, let's take 'people' variable, and I can run a function called '.append'. So, I'm doing this literally on the list. So, 'people.append( )' and inside of here, I can put anything in. I could put a string. I could put a number. I could put a list. And that will basically be added into the list. Sort of plugged in there. So, I can append the string '('Mattan')'. So, now 'people' is going to have a value inside of it and it's going to be 'Mattan' Let's also '.append('Daniel')' in there. And then let's '.append('Sam')' as well, right? So, what should the people list have now? It should basically have three different elements as we called them before.

And the elements are 'Mattan', 'Daniel', 'Sam', and the order is actually going to be kept. Now, if I wanted, I could also remove someone from the list. So, I could say 'people.remove (‘Daniel’)'. So, what do you think is going to be inside the list at this point once I have removed them? Well, let's give it a shot. Let's try just printing out people, 'print(people)', and we could run this file, 'python lists.py' and there we go. We've just printed out the people list and it looks like it has two things inside of it.

It's got ['Mattan', 'Sam'] Because Daniel was in there but then we removed him out of it. Now, one thing to note is, even though we added 'in' strings with double quotes, the list that we're getting back has single quotes of the strings, and that's because actually, Python will convert most strings into single quotes when it shows them to you. When you're printing them out, it will almost always print them out with single quotes. Again, it doesn't matter because they're the same anyway but it's just something to be aware of. It's not, you know, that big of a deal and it's nothing that I would get all that concerned about.

What are some other ways that you can create lists? We can take a string that has commas inside of it. So, let's take the string ‘New York, San Francisco, London’ Now, this looks like a list, right? To you and I we would call this a list because it's three things with commas in between them. But to Python, this is a string and it's a string because it starts with a quotation mark and then it ends with the same quotation mark. It's not a list in the sense that you can't do a lot of the things you'd want to do to a list in Python. You can't append to this. But we could turn it into a list and I'll show you how you would do that.

You could take any string in Python and turn it into a list by using the string function split. Now, by default, split will split something based on just spaces. So, if I split this into a list, it would, the first element would be New York with a comma at the end. So, what I want to do here actually is tell it I want a split based on a comma and a space. And the way I do that is inside of the () here in split, we'll put a string to tell it exactly what you want to split on. So, a comma and a space. 'split (", ")' This is one example of what's called a optional argument.

So, this function works on its own without any things put into the (), but you can add something in there to kind of change the way that it functions. So, in this case, you know I could, let's say, print out this entire thing, and you'll see now that when I run this, now I have a list with '['New York', 'San Francisco', 'London']' and that's because I 'split(", ")' Alternatively, there's a way to join a list back into a string and that looks something like this. ", ". So, I'm taking this string that I want to use to join a list back together and I do '.join' at the end and then I pass a list into this join function.

So, one example might be, let's say, pass the list '", ".join(["Milk", "Eggs", "Cheese"])'. Now, I need to print this if I actually want to see anything. So, let's wrap this with 'print' and then I can run the code and what I'll end up with is something that just looks like regular text. 'Milk, Eggs, Cheese'. So, both of these ways, one of them allows you to get from a string into a list. The other one allows you to get from a list back into a string. Now, this second one is admittedly a little bit weird and confusing because it's not the list that you do .join on, it's a string that you do .join on.

And specifically, the string is what you want to end up in between all of these elements; this ', ' and this ', ' and it doesn't go at the end because it knows that that's just the last one. And this is just something, I don't know why it's done in this order, but it's like a weird quirk of

Python that that's how they expect you to be able to do that. An interesting side note is now that we know how to append and remove stuff from a list, we can actually use this to go back to our happy hour problem and solve the challenge that we had before which was in the happy hour file, do you remember how every once in a while when we pulled a random person from our people list and then we pulled a second random person, that second random person was the same as the first random person?

Well, how might we solve this given what we now know? Pause this video and see if you can figure it out. Well, hopefully, you thought well, now that I pulled out a random person, I can actually just remove that random person from this list before picking a second random person. And in fact, that's probably the simplest way of solving this problem, just eliminate that person as an option entirely. It does, of course, take them out of the list, so that might not be ideal depending on, you know if you need that initial person in the list and you're using it later on for something. But in this case, we're not and so this should be fine. So, that covers creating lists and now we're going to talk a little more about what you can do with lists and why are they so powerful in Python.

****

### Video 9 – Accessing Elements of Lists

So, let's talk about accessing individual elements of a list. Now, let's say I called this list that's created something like cities. So, let me extract this out of the print statement, put it up here, and call this 'cities'. So, now 'cities' should actually have a list inside of it even though this is a string, but we've split the string into a list, and here we can just, 'print(cities)', again. Right, so we have cities, what if I wanted to grab just the first city? Well, you can '# Access elements of a list using [ ]'. Now, this is a little bit confusing because we've already mentioned that square brackets are used for creating lists, but they're also used for pulling things out of lists, and I'll show you how that looks.

So, if I wanted to get the first thing from a list, the first city, for example, I can say, let's create a variable called 'first_city', and we'll set it '= cities[0]'. So, the square brackets is going after the variable, which we created up here that has a list inside of it. Sorry, the 'cities', yeah, the square brackets are going after that variable, and inside of the square brackets, we're putting a number. Now, why are we putting zero if we want to get the first city? It's because in Python, and in many other programming languages, lists start at zero.

They're called, they're what's called zero-indexed, which means that the first element is at position zero, the second element is at position one and so on and so forth. Now, occasionally, this introduces problems, things called off by one errors where people miscalculate. It does happen pretty regularly, and it's one of those counter-intuitive programmer things that non-programmers are really confused about the first time they see it. So, getting the first city is done like this. What about if I wanted to get the second_city? Well, that would be 'cities[1]'. What about the 'last_city'?

Well, we could count, and we can go through and say, well if the first one is 0, and the second one is 1, then the third one would be 2, but there's another way of getting the last one regardless of knowing how long the list is, and that's by putting '-1'. So, anytime you have a list, you can always use the '-1' position to grab the last element in that list, and you could

actually go backwards from there as well. '-2' gives you the second to last, '-3' third to last, and so on and so forth. So, you can go positive direction or negative direction, which is somewhat useful.

Finally, you can actually take out a part of a list, and create a separate list from it, and I'll show you that if I wanted the 'first_two_cities', let's create a new variable for that. I can set that '= cities' and then in '[0:2]'. So, what this is doing is, it's saying give me back the first up until the third, but not including the third. So, even though these two would normally correspond to the third thing in the list, the way that this ':' thing works is, it says, give me back the first one up until the last one but not including it. So, the value of this list would end up being New York and San Francisco.

And in fact, we can 'print(first_two_cities)', run this code, and here you'll see that this list now has 'New York and San Francisco'. The technical name for it is slice notation, and the idea is you use it for slicing things out of lists and other things, and it will come up again later, especially when we're dealing with large datasets, as a way to say grab the first three rows of something or the last 10 or whatever. And there's a bunch of sort of more advanced stuff you can do here, like you can say, get the first 10 rows, but only every other one. But that's where we'll leave slice notation for now.

****

### Video 10 – Lists and For Loops

So, one of the most useful things that you can do with lists is something called a loop; to loop over the list and perform the same action multiple times. And in order to explore that, we'll create a new file this time and save this one as 'loops.py'. Loops are really one of the areas where Python shines. And we're going to use something called a "for loop". So, right '# Use for to loop over a list'. Now, we're going to use a list of just starting with the numbers 1, 2, and 3 for now. So, 'numbers = [1, 2, 3]' and the way I would loop over this is I would write the following 'for number in numbers:' 'print (number)' And I can run this code.

And you'll see the numbers 1, 2, and 3 get printed out. Now essentially, what's happening is it's running print number for every single element of this list where, you know, we're calling it 'number' at that time. So, it seems a little complex especially, for something that would be relatively easy to do manually, just print out 1, 2, and 3. But if you have a list with a lot of different things inside of it or if the list is really long, then this is an easy way to do the same thing multiple times.

This is very similar to doing something like the following 'numbers = [1, 2, 3]' and then actually saying 'number = numbers[0]' and then pulling out the first thing, remember, which is at 0, and then saying 'print (number)', and then doing the same thing with the second one, 'number = numbers[1]' and then 'print' that. And then finally, 'number = numbers[2]' numbers two, which is the last one and then 'print (number)'. These two are actually doing essentially the same thing. And you'll see that you get the same result in both cases.

It's just that a loop is a faster and sort of more concise way of doing the same thing. Essentially, the loop is running this code as many times as there are things in the list. And each time it goes through this loop, it uses this to refer to an element of the list. And that's what this 'for x in y' kind of structure lets us do. Let's do some other examples of this to sort

of help get us more familiar with this. Let's create another list, and we'll call the 'stocks' and stocks will be we'll put in Facebook, and let's do this with all lowercase, 'stocks = ["fb", "aapl", "nflx", "goog"]' Okay, so now we have stocks.

So, if we wanted to loop over this list and print out each stock but capitalized, how might we do that? Pause this video now. Try to solve this on your own. Okay, so similar to what we had above. We could do 'for', now what do we put in here? Typically, I just pick the singular word of whatever the list is so stocks. So, the single version would be stock. So, 'for stock in stocks:' Now we print out 'print(stock.upper())' and you can actually do like a string manipulation here at the run the string uppercase function. And if we run this, and we'll see FB, AAPL, NFLX, and GOOG. Now, this 'for loop' structure can be a little confusing to people the first time they see it.

The ordering, Why is it 'for stock in stocks:' and not the other way around. Well, the second one is always going to be pre-existing list, but the first one is was what, like, what is this stock thing, where did it come from? Well, it actually gets created inside of the loop, when I do 'for something' in a list. This something will exist only inside of these indentations after this colon. So this, the whole 'for' and then with a ':' at the end, we've seen something like that before when we used 'if', we put a ':' at the end and then things are tabbed in.

Now, everything that's tabbed in will get looped over and over again. And how do we refer to an individual element of a list inside of a loop? We use whatever variable we created inside of the 'for loop'. So, in this case, it's 'stock in stocks:' A lot of times programmers, you know, maybe they get a little lazy or that they don't need to write out a full word. So, they might use something like 'i' for 'i in stocks' and then you just 'print (i.upper())' Again, it really doesn't matter what you use here as long as you use the same variable name or same letter inside of the 'for loop'. But for clarity, I'd prefer to put the entire thing and print 'stock in stocks' such as that. So, that's two examples. And we're going to do a challenge in the next video.

****

### Video 11 – Challenge: Looping

So, I'm going to give you a looping challenge, and that's, I want you to print out the squares of the numbers from 1 to 100. This is more of a complex challenge than you may have gotten in the past and so I would say break it down into multiple steps. So, as a hint, first, figure out how to print out just the numbers from 1 to 100. And ideally, you're not creating a list of numbers from 1 to 100. So, think, is there a Python way to do this? Give this one a shot on your own. And then in the next video, I'll show you how to solve it.

****

### Video 12 – Looping Challenge Solution

Did you figure out how to solve this one? Alright. Let's see how I would do it. So, I'll just make a note here for the '# Looping Challenge'. Now, how did I print out the squares of any number of numbers? Well, you know, I could do this in manual way and say numbers equals 1, 2, 3, 4, 5, et cetera, and, you know, maybe even go all the way to 10 just for demonstration purposes. And I'll show you how to print out the squares of the numbers from 1 to 10. I would

say 'for', and then I'd say 'i numbers:' or 'number in numbers:' I can just print number, and then how would I do square?

Well, I can either say number times number; '(number*number)' or number to the second power; '(number**2)' both of those will work. And here, I run it and I see I'm getting 1, 4, 9, 16, and so on. So, it's working, but I don't want to go all the way to 100 manually. So, how would I do this the Python way? Well, I hope that you tried to Google it. So, if you Googled something like python list of numbers from 1 to 100, what you should find, and here's the stack Overflow page, that there's a function called range that you can use. And range will give you essentially a list of numbers between any two numbers.

So, range 11, 17 gives you 11, 12, 13, 14, 15, 16, doesn't include the last number, which is interesting, and I'll show you how that works. It says in Python 3, you need to convert it to a list as well, and there's this list function you can use to convert a range into a list. But I'll show you both, and actually, both ways will work. So, in here, I can either just say 'numbers = range( )' and then do like, let's try '(1, 100)'.

Or I can just plug this directly in there, and I don't actually have to create a whole separate variable and just put it directly into there and it's, you know, straight forward enough. Now, if I did this, so for range of 1 to 100, what you'll notice, and it does work, starts with 1 but it actually only goes up to 99, and that's because the range function goes up until but not including the last number which was a lot like how the slice notation worked from the previous example, when you got the first thing in the list up until the third, but not including it.

So, actually, if you want the numbers from 1 to 100, you need to select the range from 1 up until 101, and that will do it. And then as a result, of the last number here will be 10,000, which is 100 squared. Again, you could convert this into a list, and that would essentially do the same thing, but it's not necessary because you'll get the same answer. And even though I talked about lists, there are actually multiple kinds of things in Python that work like lists in that you could loop over them, or you can append things to them, or things like that.

Range is one of those things where it's like a list. There are other things, a tuple is something in Python that is like a list, but you can't edit it later. But I didn't want to, you know, throw all those things at you, and they're not used all that often, so they're not that important to understand all the different possible lists that exist in Python when you're first starting out. But this is how you would solve this particular challenge.

****

### Video 13 – Creating a New List out of the Squares

What are some other things you can do with 'for loops'? Well, so far, we've just printed inside of a 'for loop,' and that's it. But a really useful thing you can do with a 'for loop' is rather than just print it out, actually append it into a list. That way, we have a list of say numbers that are the squares of 1 to 100, just as an example. So, remember before that we can create an empty list and then append into that list. Well, we can do that append inside of a 'for loop'. Think of it as like inception where you have a 'for loop' and then append inside of it. So, it can get a little complicated, but we'll walk through a few examples.

So, let's say '# You can do more than just print in a for loop'. And, in fact, often, you will see multiple things in a 'for loop'. So, how would I do this? Well, first, I'm going to start out with an empty list. Just say 'squares = []' and now, I'll actually do the same 'for loop' as I have above, but rather than print it, I'll append them into the list. So, I'll say 'for number in range(1, 101):' Now, how would I append into this variable? Well remember, I can do 'squares.append' and then add something into it.

So, what do I want to append here? I want to '.append (number**2)' So, what this is doing is basically, it's running that thing 100 times, and every time it's doing 1 squared and adding that in, 2 squared and adding that in, 3 squared, and so on and so forth. Then, when we're done, what we should have is a list of squares with all of these things inside of it. So, let's try to 'print (squares)' here and see, make sure that we're getting it right. So, I save this. I'm going to run this file.

And, of course, I still see everything printed, because that's what I was doing here during the looping challenge, but here, this last list that I'm seeing printed out from 1 to 10,000, that's this part where I'm printing out (squares). So, it looks like it's working. And that's, you know, pretty convenient. Now, there's a number of ways of potentially doing this wrong, and I'll walk us through some of them, so we get a better understanding of how loops essentially work. One thing that someone might do is instead of printing out squares all the way after the loop, if they printed out squares inside of the loop, so notice there that I tabbed in this print.

As a result of tabbing, this print is now inside of this 'for loop'. It's not outside or after it. And what that means is, well, we'll see what that does when I run this file. Okay, so I'm seeing a lot of things printed out here. And you'll notice, if you sort of scroll to the top, look at this like pyramid staircase kind of structure. It's pretty. It's probably not what we wanted. What's happening here is that it's basically printing the value of squares every single loop after it adds a number in.

So, the first time the loop runs, we've added one squared into there, so this, now it's a list of 1. And the second time we've added 2 squared in, so now it's 1 and 4, and then we add 3, and so on and so forth. So, we're printing out the value of squares with every iteration, which, you know, that's what happens when you do something inside of a loop. It's going to run many, many times. So, we don't want this print inside of there. We also wouldn't want to put 'squares= []' inside of the 'for loop' either.

As you can kind of imagine, what happens at this point is that even though I keep adding stuff into this list, I also keep clearing it out every single time. So, if I run this and I print out the result of squares, it's only going to have one value in it, and that's 10,000, and that was the last thing you put into it because we cleared it out right before. Even though we've actually done all this work, we've lost of all of it. So, that's why, you know, in order to, say, do a loop, add stuff into a list, we have to do it kind of exactly this way.

****

### Video 14 – A Common Python Pattern: Creating a List from Another List

So, we can do something similar to what I showed you in the last video. For example, take this list of stocks we have here, uppercase each one, and append that into a new list of uppercased stock. It's an easy way of taking an entire list, running the same action on every

element, and it's like an apply all kind of thing. Now, I could show it to you this way, but there's also a shortcut here and it's something called a list comprehension. Now, it's not really necessary that you know how to do a list comprehension, except that it's something a lot of Python developers do a lot because coders are lazy, so they try to make things as short and as easy as possible.

So, I'll show it to you. And this is called a list comprehension, and the basic idea here is, you know, we have stocks, so I could loop over stocks, uppercase each one and append it into a new list of, say, uppercased stocks, but what if I don't want to have to create an entire new list just to append it into. The way a list comprehension works is like this. You put it inside of '[]', so list comprehensions, they look like you're creating a list, but inside of the list we're putting a loop.

So, and the order of this, it's a little weird so, you know, bear with me here. '[stock.upper() for stock in stocks]' So, that's a list comprehension. And essentially, what we've done is, we've just packed this entire thing inside of one line, and the net result is going to be a new list where we're looping over this list of stocks, which I've already created above. We're calling each one stock, and then we're saying 'stock.upper' for each one of those. In order to see what this result is, I'll have to actually print out the entire thing.

And here you'll see that the end result is this list that has '['FB', 'AAPL', 'NFLX', 'GOOG']' all uppercased, and we didn't have to give it a new name, variable name, although we could if I took this out and, you know, called it 'uppercased_stocks=' that, and then just print (uppercased_stocks) That would work the same way. Now, a list comprehension, if it's really confusing or if you're having trouble understanding it, then don't worry about it.

You can do the exact same thing just as I mentioned before. You can create a new list. You can loop over the previous list and append into it and, boom, now you've got the same result, but this is a way of doing it in less lines, and you might just see this around. This also has some other functionality. Like you can add and 'if' inside of the loop as well, but we don't need to go that deep into list comprehensions.

****

### Video 15 – Fizzbuzz Challenge Setup

So, I'm going to introduce something I think kind of as a super challenge, and it's called Fizz Buzz. Fizz Buzz, in the developer community, is an infamous problem because it's often something that interviewers ask developers to solve on the spot, sitting at the interview, and it's pretty nerve-racking to do. However, we now know enough to be able to solve this challenge on our own, so let's give it a shot. And we have as much time as we need, although I'd recommend setting aside about 10 minutes or so to see if you can figure this out. And the way this challenge works is as follows.

I want you to write a program, and meaning call it fizzbuzz.py, or whatever you want, just a Python script, and I want it to print out the numbers from 1 to 100. That's pretty easy so far. We know how to do that. However, for multiples of 3, print 'Fizz' instead of the number, and for multiples of 5, print 'Buzz'. For numbers that are both multiples of 3 and 5, print 'FizzBuzz'. Alright. So, a few things here. First of all, how do you know if a number is a multiple of something? Well, there's this thing called a modulo (%).

So, this is a math symbol that Python has and it looks like the percent sign, and what Modulo (%) does is it tells you what's left over when you divide one number by another number. So, 9 % 3 or 9 % 3 = 0; because actually 3 goes into 9 three times with no remainders. So, the % will end up being 0. However, 7% 3 is not 0; it's actually 1 because 3 goes into 7 two times with 1 left over. So, you can use the % and the two equal signs to check whether one number goes into another number evenly.

So, there's two examples that I'm showing you on this slide. 9 % 3 == 0 # True otherwise, it will be False. 7 % 3 == 0 #False So, that's a way of telling if a number is divisible by 5 or 3 or 5 and 3, et cetera. The other thing here is we know how to loop over the numbers from 1 to 100, but how do we check the value of one of those numbers. We're going to have to use 'if' statements; 'if', 'elseif', 'else', so on and so forth. So, again, try to tackle this, take it step by step and see how far you can get, and then in the next video I'll cover how I would solve FizzBuzz.

****

### Video 16 – Fizzbuzz Challenge Solution

Did you figure it out? If not, don't worry about it. It was actually a very challenging problem. So, if you figured it out, great. If not, then we'll go through it together, and you'll learn something, and that's fine too. So, I've already created a file called 'Fizzbuzz.py', and we can use that to get started. So, the challenge was print out the numbers from 1 to 100 but for, but we can leave it for there. We'll actually tackle this in steps. So, the first thing we want to just do is print out the numbers from 1 to 100, and we'll start out by, you know, just writing #Fizzbuzz challenge here at the top.

Now, did you figure out how to print out the numbers from 1 to 100, at least? Hopefully, because we already did something very similar. We'll do a 'for loop'. So, 'for number in range (1, 100):' Now, if you don't remember at this point, does range include the last number or not? Then you can just try out both of them. 'of range(1, 100):' 'print (number)' and you could run this, 'python fizzbuzz.py' and you'll see oh, it actually just goes from 1 to 99. So, I need to change this and make that '101' and run that again, and there we go. So, that's the first part of it.

It's relatively simple. Now, maybe you've thought of creating an empty list and appending the number into an empty list. That's fine also. Either way, as long as you can get this range of numbers working, doesn't really matter how you did it. But now, the first challenge is introduced. How do we use, how do we check to see if the number is divisible by three and print out Fizz if it is? Well, did you figure out that you can do an 'if' statement inside of a 'for loop'? Remember we talked about nested 'if' statements, where you can do an 'if' inside of another 'if'? Well, you can also do an 'if' inside of a 'for loop'.

So, we have this loop where we're going through the numbers from 1 to 100, and we can check to see what the value is of this number variable we've created at that loop. So, here we're going to say 'if', make sure it's tabbed in. It has to be inside of this 'for loop'. So, 'if number' and how do we check if something's divisible by something? We use the modulo symbol that I mentioned before. So, 'if number % 3 == 0:' well then we know it is divisible by 3

in which case we can print ("Fizz"). Now, let's leave this at this point and run it and see what happens because I haven't put in an else, or I haven't done anything else in here.

And if I run this code now what I'll see is, I see a bunch of Fizz's but it's not exactly what was in the challenge because what I said was, if the number's divisible by 3, print Fizz instead of the number. What we have here is 1, 2, Fizz, 3, 4, 5, Fizz, 6. So, we're still seeing the number. And that's because on line 5, we're still printing out the number in every loop. So, what we actually want is an if and an else, meaning, if the number is divisible by 3, 'print (‘Fizz’)' 'else:', 'print (number)' And this is where we want to put that print number inside of that 'else'.

That way, it's only doing one or the other. Is this starting to make sense? I'll run this now and show to you that now I'm seeing 1, 2, Fizz, 4, 5, Fizz, and so on. So, it's starting to look kind of how we want it to. Checking to see if it's divisible by 5 is just a matter of putting a second 'if' in there which we do using 'elif'. 'elif number % 5 == 0:' 'print (‘Buzz’)', okay? So, now we have these two conditions. It's either one or the other. Okay, so we're seeing some Fizz's, some Buzz's. Looks pretty good. I'm getting 1, 2, Fizz, 4, Buzz, because that's where five would be.

Alright, let's add in that final step, where if the number is both divisible by 3 and 5 'print (‘Fizzbuzz’)' So, you know, maybe I go in here and say this is a third 'elif'. And I say, okay, 'elif number % 3 == 0:' if it's divisible by 3 'and number % 5 == 0:' So, if it's divisible by 5 as well, then 'print (‘Fizzbuzz’)' Now, if you're saying this isn't going to work, well then you're right. So, I've just run the code and I'm not seeing any Fizzbuzz's in there. I'm seeing, okay, let's go down to 15.

That's the first number that should be divisible by both 3 and 5. I'm only seeing Fizz. So, why is that? Maybe you figured this out on your own or maybe this is the first time you're seeing it, in which case, pause the video and see if you can look through the code and figure out why we're not seeing this third 'if' clause being run. Well, the answer for this, as I mentioned with how 'ifs' work is it's like setting a bunch of doors and numbering them. And you can choose one of those doors to walk through. But you can only choose one door. And it goes from top to bottom.

So, actually the order of this matters which means that if a number is divisible by 3, then this first Fizz will get run, it'll never get down to the second 'elif' or the third 'elif' or the 'else', which is why the 'else' only runs if none of the ones above it end up being true. The problem with that is that in this case, if it's divisible by 3 and divisible by 5, well in both cases it's already matched one of these above checks so it's never going to get down to this third one. So, how do we deal with this issue? Well, probably the simplest way to do it is to just move this entire check up to the top.

Because it's the most specific one, if you check for that first, then only check for the other options later, you know, that's one way to do it. So, let's go ahead and grab this entire thing, cut it out, paste it up top, and we'll have to do a bit of changing of this beginning. So, the first one needs to be 'if' and then the next 'elif'. So, we're flipping around the if and elif. So, now the first thing we're doing is checking is it divisible by 3 and 5? Then 'print (‘Fizzbuzz’)'. Otherwise, check if it's divisible by 3, or check if it's divisible by 5. So, let's see if this works.

Let's clear this out, cross our fingers, run the code, and it looks like we're seeing some Fizzbuzz's in there. So, we're getting, trying to scroll up. Oh. yeah, there we go. Command line issues. So, we see 1, 2, Fizz, 4, buzz. And then if we go down to where 15 would be, I'm seeing Fizzbuzz. So, this is probably the simplest way to do it. It's also possible to not change it around and add in like an 'and', like a check to make sure it's not divisible by, that it's divisible by 3 and not divisible by 5 and vice versa. But this is probably the easiest way that requires the least amount of code.

Now, as an interesting side note, there are many different ways of solving the Fizzbuzz challenge, and if you search online you could probably see some of the craziest ways. You know just Python one line Fizzbuzz, and it's, you know, worth exploring that, for example here, you can see, well, here's one way of solving Fizzbuzz in one line. Print and notice here this print, this is definitely Python 2 because there's no parenthesis around this print function. But if you copied and pasted this in and then put parenthesis around this print, or after the print and around at the end here, this would also work and does it in one line.

It's using joins. It's using a list comprehension in here. So, actually doing this loop in just one line. It's not all that necessary for you to get that, you know, simple as they say, just one line. In fact, sometimes trying to fit everything into one line actually makes it harder to understand and more complicated. So, I actually prefer something like this better. I find it a lot easier to read. So, if you were able to get this working and you kind of understand how my solution works, then congratulations. You've solved the Fizzbuzz challenge.

****

## Video 17 – Week 2 Recap

So, this week, we covered less than we did the first week, but the topics were more complex. We talked about if statements with if, elif, and else. We went into logic in Python, and we went into lists, and looping over lists, and all of the different possible things you can do with those. And as we saw, when you start to combine them together, the result can be somewhat complicated, but you can do really cool things really fast.

So, to test out and to get a little bit of practice this week, assignment 2 is similar to week one's assignment, where I've given you these eight prompts, plus a bonus prompt in there, and you're going to be testing out the different topics, and the different skills that we covered this week. So, give this assignment a shot on your own. Just like before, write your solutions underneath the prompts with code and then submit them, and we'll see how you did.

****

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