Week 4
Video Transcripts

Video 1 (09:23): Lists

Okay, so let's look at the final piece of the basics of Python—final data types, really. The final data types, what we looked at, so far are strings, integers, floating-point numbers. These are very basic pieces of data. In Python, we can also have collections because obviously, you need arrays and vectors and those kind of things, where you can put lots of data together— identical data together, and then, access each item individually, from that. So, collections are very useful. So, we're going to look at collections, and we'll look at how to iterate over these collections. So, let's first take a look at collections. So, the most important type of collection in Python is a list. Lists are sequential, ordered, mutable collections, and I'll explain, what each of those things mean. Sequential is pretty straightforward. The idea is that the sequence of elements matters.

Now, Python doesn't care, whether it matters or not. It matters to you- the programmer. So, for example, if I'm looking at a price series, I'm looking at the prices of, say, a stock, then it matters to me that on day 1, the— on day 1, the price was '24.2'. On day 2, the price was '25.1'. On day 3, it was '23.7'. So day 0, day 1, day 2, etcetera. So, this-what this says is-there's an ordering on my data, and that's what the word sequential and ordered means, right! The data is sequential. The order matters. It matters that this price, '24.2', is before this price, '25.1'. So, that's the first point that we have to deal with in this stuff here. So, let's take a look at some examples. So, we have here a list and the way Python recognizes the list, is if it sees a square bracket at the beginning- at the end of something.

So, the moment it sees a square bracket, it says, "Hey, that's a list", as long as it's not an indexing operator. So the difference, say, between 'x' '5' and 'x' equals '5' is that, in this case, this is an index and this is a list. And contextually, Python figures that out, it says, "Hey! Wait a sec. There's an equal sign, and then square brackets, so that's a list." And, it says, "Oh! Hey, wait a second, there's an identifier, 'x', followed by this, therefore that must be a... an indexing operator." So, here we see several examples of lists. So, we've got a list of names, which is 'John', 'Jack', 'Jill', 'Joan'. A list of tickers— Apple, 'Ions', 'GE', 'DB'. A list of natural numbers '1', '2', '3', '4', '5', '6', '7', and a long list, that contains lists inside it. So, the idea here is that, a list can actually contain lists. And in Python, there's no requirement that a list contain objects of the same type.

There's no requirement that they actually be sequential in your mind. Python will treat them as sequential, but that doesn't really matter. So here, if you look at this long list, and we count the number of elements over here, we see there's an integer, one. Then, we have a list—'a', 'b', 'b', 'c', which contains a list. So, that's a second element. Then, we have the integer '43', and then we have a string, 'Too many cooks spoil the broth.' So, the long list here, actually contains only four elements, right! One of the elements, is a complex element, because it contains 'a', 'b', and—contains a list. However, the number of—as far as the long list is concerned, if you asked it, how many elements do you have? It'll say, I have four elements.

So, those are the kind of basic examples of lists. So, let's look on and see what kind of operations we can do on lists. And, for this, let's go and see them in practice. So, if you want to create a list, we can create lists in multiple ways. We could just create a list by specifying the values. We could create a list by giving the list— calling the list function, what this says, create an empty list. Or, we can create an empty list by just specifying the square brackets, and we will get an empty list that way, either way will work. So, we get this stuff, and note now, we have a list here that has four elements—that's 'x', and a list here that has no elements, that's 'y'. We can add items to a list. The way to add items to a list is, to use a function called, 'append'.

There are other ways. We'll see them briefly. But, the append is the most common function. And what append does is, it takes an item and adds it to the back of the list. So an example here, we start with an empty list, and then we 'append' a 'one', and then we 'append' a 'two'. So, we get—we start with an empty list. We append a 'one' and we get a list with 'one' in it, and then, we append a 'two', and we get a list with 'one' and 'two' in it. So append—notice that—just make sure that, you always remember, an append takes only one thing inside it. The append function has only one argument, and whatever that value of that argument is that will get appended.

So if you append, for example, if you are trying to append a list, to this thing, if I did, 'x' equals '2', comma '3', and I did, 'x' dot 'append' '4', what's going to happen? Well, it's going to say, "Hey, I have in a list with two and three in it, two elements, and I'm appending a new element, and that element I'm appending is the list four." So, I get—I should actually print that, I get a list that contains '2', '3', and the list '4', okay! So, you've to watch out for that. So, it takes only one argument, and that argument will be as-is appended to the back of the list. You can insert too. Inserting is not usually a great idea though, it doesn't matter too much in lists. So, if I have the list 'x', which we now know contains '2', '3', and the list '4', and I want to inverse, insert the...the word, 'Half', in the zeroth place, then what's going to happen is, that currently in the zeroth part of my list, I have a '2'. So, everything's going to move up, and I'm going to get a list, that contains, 'Half' and then '2', '3', and the list '4'.

I can also extend the list, extending the list is slightly different. When you extend a list, you're going to give— the argument you're going to give is another list. And, what Python will do is, it will unpack the list, and add the items to the back of the list that you're extending. And so, here we have a list 'x', which is empty. And, we extend it by a list '1', '2', '3'. So rather than getting the list '1', '2', '3' inside it, we get the entire list contains '1', '2', '3'. Notice the difference between this and append. If I did 'x' equals list and 'x' dot 'append', '1', '2', '3', it's going to be slightly different, right! Ready? So here, what I'm getting, the list that contains one element, which is the list '1', '2', '3'. Here, I'm getting a list that contains three elements, the integers '1', '2', '3'. That's the difference, because in the case of extend, it unpacked the elements. In the case of append, it just added, the whole object itself, into the back of the list.

Since it's sequential, just like in strings, we can index and slice them. So this is, you know, exactly like strings. If you take the length of 'x', it's going to count the number of elements. If you take 'x', and say, give me element number '3', it's going to give you element number '3', which is-oops-'5'. And if you take—slice it by '2' colon '5', just like in strings, you're going to get the elements in location '2', that is the '2', location '3', location '4', but not in location '5'. So you get '2', '5', and '3'. And then, if you do 'x' negative one, it goes to the back and does that, gives you the last element, '32', and of course, you can inverse the list, by doing, 'x' colon colon negative one, just like we did with strings before. We can

remove items from a list, and there are two ways of doing that. You can use, pop, to remove a specific location, from an item of a specific location. And you can use, remove, to remove an item with a specific value from the list.
So, here we have 'x' dot 'pop', and what this will do is, it will remove the last item, '32', from the list. 'x' dot 'pop' '3' will remove whatever item is in location '3', which in this case, is the number '5' — '0', '1', '2', '3', location '5'. And then, 'x' dot 'remove' '7', will remove the first seven that it finds. In this case we have only one seven, but, if you had more than seven, it will just remove the first seven. So we get in the end, what we get is, the '32' removed, followed by the '5' removed. This '5' gone and then the '7' gone, right! Those '3' things there. As with everything, you have to be careful that you don't try to remove something that is not in the list or access something that is not in the list. So, if I wanted to remove the number '20' from 'x' and I do that, I get a value error, because '20' is not there. If I try to pop the twentieth location, I'm going to get a value error—index error, because there is no location '20' in the list.

Video 2 (10:18): Mutability

The next thing what lists is, is that they're mutable, so, let's take a look at what this means. And, this is a very important Python concept, so pay attention. The ideal mutability, we talked about this before as well a little bit, is that the contents of, whatever thing, you're pointing to can be changed in place. So, what this means is that if we have a list, 'x' equals '1', '2', '3', '4', and we do 'x' '0' equals '8', then what's gonna happen is, that this '1' over here, is going to be replaced by the '8'. So, it's changeable. And, the key in understanding mutability and immutability is that immutable object is that, data objects cannot be changed. And in Python, the idea is that, if you have the number '5', it's always going to be the number '5', it's never going to change. If you have number 64.3, it is always going to be that. If you have a character, H, it's always going to be the character H.
They can't be changed. So, they're immutable, So, we don't allow them to be changed. On the other hand, there are things that can be changed. If I have a list of students who are enrolled in this class, and somebody drops out, I still have a list of students enrolled in this class, except that its value has changed, so that list should be modifiable, people are going to drop out, people are going to add in, and the list is going to change, so it should be modifiable, therefore, that's mutable. And, that's the basic idea–and this mutability and immutability then go on. Let's try, a few examples over here and see what happens. It is an interesting thing in Python, and, we see that it has consequences–especially with function calls–and various kinds of things. But, let's take a few examples and see. So here we have, in our first example, we have, 'y' equals 'a' and 'b'. It's a list containing two elements, the character 'a', and the character 'b'. And we have a list, 'x', that contains three elements. It contains the integer '1', it contains a list, 'y', and the integer '3'.
When you say a list 'y', what we mean is, it's the value of 'y', that goes in over there. So if I print 'x', and I print 'y', then I'm going to see that 'x' will be—and let me do that–here. So, let's do this. So we see that 'x' actually is '1', the list, 'a', 'b', and '3', and 'y' is 'a' and 'b'. Okay! So now, let's see what happens, if we change the value of 'y', because it's a list, it's mutable, so I can replace 'y' '1', which is the character 'b', with the number '4', with integer '4', and then I print 'y'. So, 'y' now becomes the character 'a', as in the

first element, the zeroth place, and the integer '4', in the oneth place, of this thing. What happens to 'x', thought? Because 'x', we saw, had, when we looked at it, it had these values, right? It had '1', a list, 'a', 'b', and '3'. But this list 'a', 'b', is actually the list 'y', right! It's not a list 'a' 'b', it's the list 'y', because this is what we give it. So, this is what 'y' is, and we change 'y', so what happens is, that 'x' will also change. Let me see, that's right. 'x' is now, contains 'a' '4' rather than 'a' 'b', and the reason is, because the address in memory, of where 'y' is, and where the middle part of 'x' is, what's in location one of 'x' is–is the same location. So if you change that, it's going to change in both. Okay, that's the idea there. So, that's what mutability really buys us. Let's take a look at more examples, and...and here what we'll do, is we'll look at, the addresses and how they change. So let's say, we have a string, 'Hello', Remember, strings are not mutable, okay, they're immutable. And, I look at the value of the string itself, and its location, and then I add to that, the word 'You', okay! So, I do 'x' plus equals 'You', I'm doing an augmented assignment. And then, I print, 'x' and 'id' 'x'. So, that's for the string part, right? So, that's what this does, over here. Then, I have a list, that contains 'Hello' and I look at the address of that list, and then I add, extend the list by putting a 'You' at the end, and this plus equal to, is similar to the extend function, it's going to extend the list, and then I print that. So let's see, what's the difference between these two situations is.

So in the first case, 'Hello' was at this location. When I added the 'You', the location changed, essentially, what we've got is a new string. So the old 'Hello' is still at wherever it was, and the new 'Hello You' is in a different location. So, we haven't actually changed the string, what we've done is, we've created a new string, and we are pointing our variable at that new string. That's what this is telling us. From this value, we're now pointing at this address. In the case, remember, the 'id' function gives you the location in memory of a variable. In the second case, we have a list 'Hello', a list that contains element 'Hello', and that's at this location. Then, we extend it, by adding the word-a list, the elements of a list that contains the word 'You!' in it. So now, we get a list here, that says 'Hello', 'You!', but look at where it is, it's in exactly the same location, and that's, because a list is mutable. So, when we say it's mutable, it changes in place, whereas a string is not mutable, so if you want to change it, you have to create a new string to change it, that's the basic difference.

So, this can be kind of confusing and it has certain issues. The fact that some things are mutable, and some things are not mutable has certain effects on the way the program runs, and so here, let's take a— it's a very quintessential example of, what they call, gotchas in Python. So, here is a gotcha, what it says is, we have a function called 'eggs', and 'eggs' has two arguments, it takes something, and an item, a variable called item, and it has another something, which defaults to the value zero, right! And what we do is, we add to total, the value of item. Right! So if we call, 'eggs' without specifying total, if we call one argument, then total will default to zero, and we return total. And, we have a second function called 'spam', which takes an element, some element, and it take a list, instead of an integer, it takes a list as a default second argument.

So, the second argument is 'some_list', and it's going to default to the empty list. And then, here we do some list, dot append 'elem', so we append that, and return 'some_list'. So, what's going to happen here is, that if we call the function with just 'elem', and if we call 'spam', with only the value of 'elem', then 'some_list' should default to an empty list. So, let's see what happens over here. So first, we're going to call 'eggs' with the argument '1'. So what should happen is, item should be one, total is zero, so total

plus equals item should return '1', right! And then, we call eggs with '2' and again, we should see the same thing. So, we just commence. So, we can see the effect of the integer default, and oops, I have to run that, right! So, I didn't run that. So what we get here is, when I call 'eggs' with '1', then item is one, total is zero, so total is one plus zero, and we return total, and we get a '1' printed, then I call 'eggs' with '2', item is two, total defaults to zero, and two plus zero is two, so I return two, and I get a '2', right! So, that is how that works.

Now, let's see what happens if I try this with 'spam' instead. So, I'm going to call 'spam' with '1', so now, 'elem' is one, and 'some_list' is an empty list, and I run that, here. So here something odd happens, right! So when I call spam, with '1', then I get '1' back, which is what we expect, because 'some_list' is an empty list. So, it's going to default to an empty list. We're going to append the number '1' to the empty list, and we get a list containing '1', that's great! When I call the second time however, rather than 'some_list', I'm still not specifying 'some_list', I'm saying, "I'm calling it now with 'spam' '2', right!" So, I'm not specifying the value of 'some_list', so you would assume that 'some_list' will default to the empty list, but it doesn't. It defaults to what the previous value of 'some_list' was, because that's now been defined.

And, what is happening now is that, and I'll explain this in a second again, but what's happening now is that, instead of 'some_list', being the empty list, it becomes the list '1', because we've already created it, and it already has a value. And, when I append something to that, I get the list '1', '2'. So when we...when we do this here, and call some— call the function 'spam', then the first time we call the function, Python's going to set up a little name space for 'spam', this is going to say, "Here's spam, and this contains 'elem', and 'some_list', all right!" And 'some_list', is going to now point to a list somewhere. And, that list will be whatever it is at the end of this thing here. The second time I call it, it says, "Hey! I already have this definition over here, I have this definition here.

So I'm going to use that, and this definition, I have 'elem', which is coming from the function call, so 'elem' is coming from the function call, and I have a 'some_list' that points to a list, so I'm going to use that, because that's already there." This list contains a '1', so that's already there, right! So now, it says, "Okay, I'm going to use that." And you say, 'append', it appends to this. And, that's why it will keep getting appended to that, whereas, the total, because total is a immutable object, each time it creates a new total, because you can't change the existing total, it's going to create a new total, that's the difference between the two. So, that's mutability and it's kind of, an important Python concept, so well worth getting your head around that, so I would play around with a few examples, and check what that does here.

Video 3 (08:53): Iteration

So, let's now move on to the next type of data type that's lists. Lists are important. We'll use a lot of those. But, there are also something called, tuples, and really the only difference between a tuple and a list is essentially that a tuple is immutable. Tuples are...are also sequential ordered, which means that we can access them by indexing, and you know, getting the values the way we did with this. What we can't do is, we can't do an assignment on it, because you can't change what's inside a tuple. Tuples are immutable, that's the idea there, so that's for tuples. So now, we have lists, we have tuples. These are

sequential, ordered collections, and you might want to move through them one by one, extracting elements and working our way through them to do stuff. For example, we might want to find something in a list. We look through each element, and we find some element that matches whatever we're searching for, then we extract it and we...we are done, right! So, that's a typical—searching through a list and a typical exercise that we might want to do in...inside lists. So, let's see how we can iterate.
The iteration is fairly straightforward. There are—there's a 'for' and a 'while'. We are only going to look at 'for' right now. We'll look at 'while' when we are using it. So for now, we'll just look at the 'for' loop. And then, in the case of Python, there are two ways in which you can iterate through a list or tuple or a collection. You can use, what's called the 'range' method. And in the 'range' method, what we're doing is we specify a collection of indices, and we're going to draw from this collection of indices sequentially to get values. So, when you think of a list here, any list is really a collection of locations, right, zero, one, two, all the way to n. So, if we can iterate through these indices, by first getting a zero, then getting a one, then we can do 'x' '0', 'x' '1', or as—more generally 'x' 'i', and get values out of our list, right!
So, what we want to do is, we want to get successive values of 'i', and the 'range' command does that for us. What it does is, it pulls the successive values of 'i' all the way from zero, up to the length of prices up to this. So, the argument there is length of prices, so up to there it will pull out the values, one by one from that. And, we can see that what this is going is, it's going—we— to get the actual value itself, we need to give the name of the list that we are looking through or the name of the tuple, and then we use the indexing method, to extract values from that. So, that's the basic way of iterating. The second thing we can do is, we can access items, sequentially, by going directly to the items itself. So, we have a list here called 'prices'. And, what we say is that for each element of the list, pull an element out, and assign it to the variable 'stock_price'. So, 'stock_price' is going to be first— it's going to be 'AAPL', '96.43', then it's going to be 'IONS', '39.28', and then 'GS', '159.53'. In turn, it's going to pull each one of them out, and give it a value. So, the difference between the two is very straightforward. In this case, we need to iterate through the indices, and that's what this is. And, to extract values, we need to do 'x' of 'i'. And, in this case, what we need to do is, we iterate through values itself. So, we could say 'for thing in x', and each value of 'x' will become a value of thing, in turn, and then we can do whatever we want with it. So, that's the basics of iteration. And, you can control the iteration using a 'break' and 'else' and there's also a continue, we'll look at that in—when we're actually writing programs.
So, inside of 'for' block, you can use a 'break', to exit the block, prematurely. So, when you think of a...a 'for' loop, and you're iterating through a collection of some sort, let's say the collection has ten items, you can say, "Hey, the iteration will end normally, if I've gone through all ten items." That's a normal iteration. However, there may be times when you want to get out of the item early, like, for example, let's say you're iterating through the inventory of Walmart, right! The inventory of Walmart is millions of items or the inventory of Amazon, which is billions of items, right! So, you want to iterate through all this stuff and you're looking for a particular item, and the very first item you find, matches. You don't want to go and start looking at the other one billion minus one items, just for no reason at all, right! You already found what you wanted, you want to exit from it. So, what you would do is you would say, something like this, 'for item in prices'—so prices is our list here, and we are searching for this ticker. If 'item' '0'—so each item in turn is this tuple, right, so 'item' '0' is the ticker in the tuple, and it says, if 'item' '0' that is the ticker in the tuple, matches the ticker that we've input, then print it. Print the value

of the second thing, the '96.43'. For example, if it were Apple, and we found our item, so break out of it. What happens with the break now is that execution, when you hit the break, is going to go from here and go to this statement here, the 'for' is ended. So, that's how the break works. It just breaks out of the loop, and goes to whatever the next statement after the 'for' block is. So, you can think of the for as being one compound statement that's this entire block. If, however, we go through all the prices in this loop or all the items in Amazon, all the billion items in Amazon, and we don't find what we're looking for, which means that we've gone through everything and we haven't—this condition over here has never been true. So, we've never hit the break, then—but we finished prices completely, then what's going to happen is that the control will transfer to the 'else' part. And, in the 'else' part, we're saying print, '"Sorry",ticker, "was not found in my database"', and it does that, and prints that out.

So, the idea here is that...that... that the 'else' will occur, if the loop never hits a break. So, it's a sort of default condition, right! So, you're looking through a collection of some sort. You're looking through your backpack to find your calculator or your iPhone. You take each item out, one by one, and you don't find it. So, what do you do? You—you know, I guess you'll find a payphone if you're looking for your iPhone, right! Alternatively, you find your iPhone. Now, you don't want to pull out notebooks and wallets and, you know, those six-month-old bagels that you left inside your backpack just because you have started doing it, so you're going to finish it, right! You already found the iPhone, you're not going to look through the rest of the stuff, which case, you hit the break. If you don't find it, you hit the 'else' and go to the payphone, that's the idea there. So, that's the...the basics in 'for'. So, let's go and take a look at how this works, in practice.

So, let's see here. We've got a...a basic 'for' here, we have a list 'x', which contains these numbers, and we want to index—index to take the...the indices zero through whatever the length of this list is, and that's what we get here, right! So what this is going to do is— and we can actually do this here, 'index' comma 'x' 'index', and it tells me index zero has a value one, index one has a value seven, index two has a value two, etcetera...etcetera. So, that's a simple one, and we can see what 'range' is doing is, all it's really doing is generating from zero up to the length of 'x', is generating a series of numbers, it's actually—it's called a generator. It doesn't actually do— create this list, which is why, I'm forcing it to create a list here. It creates some—you can think of it as some kind of machine that it's going to throw out numbers one by one, and the idea is simple. You're searching through Amazon. You have a billion items.

You don't want to create a second list of size one billion, just to hold indices, right! You'd rather have a little machine that throws indices out one by one, so that you're not wasting space. That's the...the goal there, so that's that. And, the second method is, of course, to iterate through the list by pulling each element out in turn. So, what this says is we have a list 'x' here, the same list, and now rather than putting out indices, we're going to pull the actual elements out itselves, themselves. So, that's what we get here. So, each element is an element. So, notice the difference. Here, we're printing element, and here we're printing 'x' 'index', okay. So, we have to refer to 'x', explicitly, inside our loop, here. Once, we've put an 'x' over here, there is no reference to 'x' inside the loop anymore. We don't really need that anymore.

Video 4 (07:04): Example

So, let's do a simple problem here. We're going to write a function 'search_list' that searches the list of tuple pairs and returns the value associated with the first element of that pair. You might want to try this yourself first, and you can pause the video here, and come back later and try it, but what I want to do is, I want to work through this function so you get a sense for, how we write functions and how we iterate and, you know, all that kind of stuff, putting it altogether, so to speak. So, what do we do here? So–so if you want to pause, pause and come back. Okay, I'm assuming you're back. So here, let's say we've got a list of prices that are, kind of like this, we have a list of prices, each price is a tuple with the ticker, and the actual price in it. And then, we've got a ticker that we are searching for, so we're going to search for 'IONS', for example, and we want to call our 'search_list' function that searches for the price ticker combination.

So, what it's going to do is it's going to look– iterate through our list and find the tuple that matches– the first item of the tuple that matches ticker, and once it finds that it's going to return the price that is associated with the ticker. So, for Apple it'll...it'll return '96.43' or maybe the entire tuple, it...it's up to us, whatever we want to do. So, let's see how that will work. So, first thing, I've already written the list of tuple value stuff over there, so we already have our function declaration, so here we've got the word-keyword 'def' that says it's a function declaration definition that follows. We have the name of the function, 'search_list', and we have the two arguments, a 'list_of_tuples' and a 'value', right! So, that's what we want to work with. So, 'list_of_tuples' is our first argument and 'value' is what we're searching for, and so we want to search for this stuff here, we could say 'for thing' or let's use a nicer word 'for element in list_of_tuples', and notice I can always press the tab key. I mean, you don't notice that you can't see it, but if I press the tab key, it tries to fill up, whatever I've done over there.

For element and list of tuples–and I want to compare whether the first item in element matches the value, right, so if 'element' '0' equals 'value', and if it does, I want to return the value, right, so I can say 'return value'. Okay! So that's the first thing I can do. So, this is actually going to give me the value for any element that is– let's test this out, okay! So, if I do this and I do this, it returns value. Oh sorry! I shouldn't be returning value, I should be returning–see this is how we all make mistakes, when we write program, I should be returning 'element' '1', because value is a 'ticker', right, so that we'll keep that in, so that, you know that we can all make mistakes. So, I do this and I get '39.28', which is the value associated with 'IONS' in my tuple over there. So, that's the first step. But, of course, what happens now, if I instead of this– instead of 'IONS', I'd look for, say, Google, I get 'None' back.

The reason I get 'None' back is because my function hasn't really learned– hasn't returned anything. So, what I would like to do instead, is return say zero, right! It just depends on what I want. Maybe I'm happy with 'None', but let's say, I want to return zero. So, what I can do here is I could use the 'else' part of 'for' and say 'else' 'return' '0', right! So, if I do 'else' 'return' '0', and now I search for 'GOOG', I get '0', right! The-notice a couple of things here, one is that, I don't need to use the break anymore. I don't need the break, because when I 'return' 'element' '1', it's going to end my function immediately. So, one of the things about functions is that the moment a function hits a 'return' statement the function ends. So, if the function ends, it's not going to look at anything else and we can check that out. So, if I look here and I print the value of element 'print' 'element' to see which ones we are considering, right, and if I search for 'GOOG', I'm searching through the entire list. I get an element for 'AAPL', I get 'IONS'. I get 'GS', okay! However, if I search for 'AAPL', then I only look at 'AAPL' because the return is going to end

that, and what this also means is that I don't really need an 'else' either because, in this case, what's going to happen is that since the function will end, the moment I find my element, and it hits the return statement, the only way I'm going to get to anything beyond the 'for' is, if I don't find the element. So, I don't need an 'else'. I can just take this, and do a 'return' '0' outside the 'for', right, not inside the 'for', big difference, right! If I do inside the 'for', what'll happen? Let's check that. Okay! So, if I do it inside the 'for'—and now if I search for 'AAPL' that works fine, but if I do it here and I search for 'IONS', okay, what...what's going to happen now let's see. I don't get the value. The reason I don't get the value is because my return is here, and this is really important because indenting is crucial in Python—but the return is here, the return is inside the 'for'. So, what it's going to do is it's going to print 'AAPL' '96.43', that's what this is going to do. It's going to check wether AAPL equals IONS. It doesn't, so that fails, so it returns zero and...and the program ends, right, the function ends. So, I don't want that, what I want is this over here after the 'for'. So now, if I do this, I look for 'IONS', we get 'IONS', and if I do... if I do 'GOOG', then I get all three.

So, what I want to do now, of course, is to remove the 'print' from here because I don't want my function, and the general rule, you don't want your function to do any printing. The reason is, that a function is like an assistant, right! You don't want the assistant to do anything important, you want to have control over how your program interacts with the outside world. So, if a function is printing something, then it's interacting with the outside world and you have no control over it, which is not a good thing, right! So, you want your functions to generally return stuff, but not print or do anything that interacts with the outside world, not get inputs, not print, unless they're designed for printing or getting input, which is a different story, like a function that format— does a formatted print or something, that's different. But, in general, you want to avoid that so try not to put 'print' statement inside a function, except when you're testing of course. So, we do this, and for 'GOOG' we get a '0' there, right! So, that's the idea here.

Video 5 (10:03): Dictionaries and sets

So, let's go to the last bit of our Python basics and, that is, looking at dictionaries. Dictionaries are very important because a lot of the data that you get on the...the Internet through APIs will come back in the form of a JSON dictionary or a...a list that contains dictionaries, or dictionaries that contain dictionaries. So, dictionaries are kind of important for data work, right, and there are many—No...NoSQL databases that also use dictionaries. So, it's sort of a nice thing to know, but they are actually very simple. The whole idea in a dictionary is that you have a pair of...of elements, there's a key, sorry, there's a key and a value. And, if you think about it, it's just like a physical dictionary in—not just like, but similar to that. In a physical dictionary, you have words and their meanings, and the way you access things in the dictionary is that you take a word, like you look for the word, 'Python' and so you'll go through the dictionary, you'll look for 'Python' and you find 'Python', and you find its definition, a reticulated snake that lives in Burma or something like that, and that's how you access the definition of python. What you don't do, in a dictionary is you don't say, "Hey, I want to look for a reticulated snake that lives in python— sorry, a reticulated snake that lives in Burma", and find the word associated with that. That would be crazy, you'll have to go through every definition, in, you know, this massive one-thousand-

page book, and until you find what you're looking for, and if the wording is slightly different, you may never find it. So, in–typically in dictionaries and real dictionaries and Python dictionaries, you access values through the key. So, the key is for access and it gives you a value right. That's the general idea. The thing about keys is they have to be immutable, you can't put mutable objects in a key, so you can't have a list as a key.

The reason is that the keys are actually mapped to locations using a function, and if the value changes, then you're going to lose its location, it's a hashed function, right! So, the key has to be a immutable object, the value can be anything, it doesn't matter. So, here are some examples. We've got a dictionary that contains market caps of stocks, there's a ticker 'AAPL', and the market cap '538.7'—it's not that anymore it's much higher, the ticker 'GOOG', '68.7' much much higher, and 'IONS', '4.6'. I just made this up, they're all—all of them are actually higher in the last few months. So, these are likely to be higher than this, I'm not sure 'IONS'. So, these are our market caps—and so the idea now is that we want to find a market cap, we're going to access it through the ticker. So, what do we do?

We say market caps, the name of the variable, and name of the dictionary identifier, and we give a ticker here, and that gives us the market cap for 'AAPL', it will return '538.7'. If we try market caps of 'GE', we get an error, because there is no 'GE' in our dictionary here. So, when you...when you want to access something, the key has to be in the dictionary, and we see in a way—let me go back actually to our example, and we can see how this works in practice. So, here we have our dictionary of market caps, if you do market caps of 'AAPL', it's going to return—and notice how the dictionary is formulated, right, there's a curly brace and a curly brace, so dictionaries are identified in Python, by an equal to sign followed by a curly brace, and knows—Python knows now it's going to be a dictionary or a set, and we'll see sets in a second, and then, it says, "Hey, I'm getting a dictionary or a set", and then, it decides whether it's a dictionary or a set based on what's inside, and inside if it finds a key followed by a colon followed by a value, that's a key-value pair, then it says that must be a dictionary, and it becomes a dictionary.

Okay, that's the idea here. So, if we look at the type of market caps, for example, it tells it's a dictionary right. So, market cap of 'AAPL' we get '538.7', market cap of 'GS', it's Goldman Sachs, an error, because Goldman Sachs is not in our—it's not a key in our dictionary. But, you can alternatively use the function dot 'get' and what dot 'get' does is, it'll return none, if Goldman Sachs or GS is not in market caps. If it is, it'll return the value, that is in the market caps. So, if I do 'mrktcaps' 'GS' equals '88.65', and here's another thing we can see actually, so if I do this here, let me do this here. So here, the one thing about dictionaries is they're not ordered. So here, we're actually seeing them in the order that I entered them, right. But, that is not necessary...that is not necessary, because the way dictionaries are stored it takes these—the values of the key and hashes them, it's using a function.

So, depending on how the hash function works out, you might get an order here. In this case, I guess what's happening is the hash function is retaining the alphabetical ordering on the keys, so that's what we are really seeing, but that's not essential. So, if we go back here, and we add—and if you want to add a new key to the dictionary, all you need to do is give the name of the dictionary, inside the index operator give the key and equal to, and give it a value. So, this now will add the key to our dictionary, and we get a new list over here, that contains Apple, Google, 'IONS', and Goldman Sachs, okay! And then, if you want to remove something, then you can just remove it by saying, "We have GOOG, yup!",

delete market caps, and we delete 'GOOG', and we get this new list over there. We can get all the keys in the dictionary by doing market cap dot 'keys', and that gives you an object back that contains the keys, it's also a generator object, so it's worth thinking that it's not actually going to give you the keys, but it's going to generate the keys, in no particular order.

If you want to sort them, what you would need to do is, you would need to say, 'sorted', notice the function right, so we're calling that, and we get them in sorted order, right! And so, these are sorted order of the keys. And, if you want to get values, you can use market capital dot 'values', if you want to get just all the values, which is not what you would normally do in a dictionary, but you can do it,in...in this over here. So, normally what you would end up doing is, you would say "Okay, 'for key in mrktcaps' 'print' 'mrktcaps' 'key'", and we get all our values over there, or 'print' 'key' comma market cap 'keys' that kind of stuff. So, this is how we iterate through a dictionary. So, we'll see a lot of dictionaries down the road, so, just as long as we are aware that they exist, and this is the basic stuff you can do, then we should be in good shape, okay!

So, the last thing that we need to worry about here is something called, a set. Sets are unordered collections, just like dictionaries, and they're a collection of unique objects. So, in a set, you're not going to have duplicate values at all. In dictionaries, lists— in dictionaries you can't have duplicate keys, in lists and tuples, you can have duplicate values, okay, but in dictionaries you can't have duplicate keys. In tickers—in sets, you can't have duplicate values. They're not going to be allowed. So, if we have a set, the way a set is recognized again is curly braces, but it doesn't contain key-value pairs, so Python would recognize it as a set. And here, we have a simple examples. I'll just walk through them. We've got a bunch of tickers, we've got a bunch of regions, these are possible sets. You can do normal set operations, so if you do '"AAPL" in tickers', if it is—'AAPL' actually isn't tickers, it'll return true, if it is not, it'll return false. So, '"AAPL" in tickers' will return true. '"IBM" not in tickers' will also return true, because 'IBM' is not in this set over here, right, there's no 'IBM' in that.

I have another set, called 'pharma_tickers' 'IONS' and 'IMCL', and I want to see whether the two sets are disjoint. In other words, is the intersection between the two sets empty or not. So, in this case, we can see that, 'IONS' is in both of them, right, we have 'IONS' here and we have 'IONS' here, right. So, the intersection is non-empty, so this is going to return false. If I do—whether they—are they disjoint or not, because the intersection has 'IONS'. We can do—check whether something is a subset by using less than equal to. We can check whether it's a proper subset. Proper subset, remember, is one that you should know is one that has—it's every element in 'pharma_tickers' is also in tickers, but there is some other element in 'tickers', that is not in 'pharma_tickers', in that case, it's a proper subset, sorry, looking at proper subsets, right, so 'tickers' is going to be greater than–more... more than 'pharma_tickers'. You can do superset, which is the opposite of that. You can do intersection, which, in this case, if we do intersection of 'tickers' and 'pharma_tickers', we're going to get 'IONS', because that's the only one that is common, right, this one, between the two. You can do union, then that's going to be 'AAPL', 'GE', 'NFLX', 'IONS', and 'IMCL' that's it, okay! You can do a set difference, so set difference, what it does is, if you do 'tickers' minus 'pharma_tickers", it's going to take out anything that is in...that is in 'pharma_tickers' from 'tickers'. So, in this case, 'IONS' will get removed and we will get 'AAPL', 'GE', 'NFLX' as our set difference.

Video 6 (10:29): Datetime library 1

So, to wind up our review of Python, we'll take a look at a library called a 'datetime' library, which is a very useful library for data analysis, and mainly because, you know, time is a of very important data element. And, the key things in understanding time is that time itself is linear. So, if you look at this here, we say time is a linear thing. It progresses all the way from, I guess, roughly from the Big Bang, when time began, and it's going to go all the way to now, and then into the future. But it keeps moving, you know, time never stops. So, once we have that, we know—we can start reasoning about time and start saying, "Hey, you know, sometime is before other time, or there's a time difference, or stuff like that", right! Because you have a linear line, which is somewhat similar to the number line. It's also important in data analysis, and the reason it's important is that data is often time-stamped. So, for example, if you're looking at financial time series data, then time is important. You want to know that a certain price is before a certain other price or a certain stock event happened at a certain time, and then the trajectory change, like a stock split, or a dividend or whatever.

You might want to look at passenger flows, by the time of the day. For example, if you're a...a...a commuter railroad planner or a commuter transportation planner, you want to know what times of the day are busy times, what is the traffic flow, which stations are busier at what points of time, etcetera, so what you do is, you get data that is time-stamped, you know, passenger flows, time-stamped, and you want to use that time. Another important thing is understanding web traffic by time of day. So, you have web traffic coming in, and you've got the time of day of...of when your traffic is higher, when it's lower. You want to look at the demographics at different points of time, so that you can show different advertisements, or show different pages, or, you know, whatever. But, the time is kind of important there too. And, of course, if you are a department store, or any, selling anything at all, then seasonality is very important, so you want to know, whether it's summer, winter, autumn, fall, January, February, March, whatever. So, time is an extremely important element. And, we see in the examples that we take down the road in this class, in this course, that often, we will be anchoring our data by a time—time-stamp. So, we need to be able to talk about time.

Python has a very nice library called a 'datetime' library that helps us talk about time. It tells us—it helps us understand the relationship between different points of time, and understands how to do operations on time. How do you find time differences and stuff like that? So, let's take a simple example—and why we need... why do we need this 'datetime' library, that's what we really want to know. So, let's see a simple example, which says, which is greater of these two dates, here. You got October 24th, 2017 or November 24th, 2016. Now, we know, looking at this that October 24th, 2017 came after November 24th, 2016. If we represent the time as strings, and then look at the ordering, which one is greater, then what we get is, apparently November 24th is greater, or after in ordering, because remember, greater than is just simply an ordering relationship, after the October 24th, 2017, which, of course, doesn't make any sense. And, this is, if you want to find out how much time has passed, we try doing 'd1' minus 'd2', and we get a type error, because you can't subtract two strings. Now, this is pretty obvious, because what we are doing is we are representing time as strings. And, as far as Python is concerned, these are just two strings. So, when it orders strings, it does them alphabetically or numerically in this case, so it says, one-zero comes before one-one. So, since one-zero comes before one-one, then irrespective of what

the year value is, you're going to get one-zero going before one-one, because that's the way alphabetical ordering works. And obviously, we can't do string operations negative—subtraction on strings, so that's not going to work.

So instead, we can try using the 'datetime' library, which is the library that Python provides us, so let's quickly take a look at what that would do for us. So, first of all, it's a library. So, when we have libraries, we need to import them. So, the first thing we do is, we import the 'datetime' library and then, from the 'datetime' library, we call this constructor, which creates a date object, and we give it our parameters, which are '2016', '11', '24', and '2017', '10', '24', and we create two variables, 'd1' and 'd2', that contain date objects. And now, if we do 'max(d1,d2)', we can see that it's correctly figured out that '2017' came after, '2016', therefore, 'd2' is greater than 'd1'. Note that here we are actually entering the data as numbers, but we can, and we'll see later, we could enter the data as a string, like we did with our earlier example, and that would also—and...and convert that into a date object, and that would also work. So, that obviously understands what is time, and we can even do subtraction on that.

So, we can see what is the difference of the duration between 'd2' and 'd1', and we run this, and we say it's '300'— we find that it's '334' days. So, what it's doing is, it's...it's actually calculating a time duration for us. So, the nice thing is that 'datetime' objects understand time, and we can do operations on that.

Take a deeper look at the library, if we look at this here, we find that there are four different objects, actually, some others we will see later, but, the four basic objects that we are— or data types that we are interested in, there is a 'date' data type, which is what we've already seen, which stores the date as a...as a form of whatever the month, the day and the year is. There is a 'time' data type, which stores the time in hours, minutes and seconds. There is a 'datetime', which stores both the date as well as the time.

So we get month, day, year, hours, minutes, and seconds, all of them in the same object. And then, there's the 'timedelta', which is what we see over here, which tells us, what is the duration, between two date time or date objects that tell us how much time has elapsed between those two objects. So let's take a look at each of these, in turn, and we find these are— we have, you know, examples for all of them. So, let's say, we start with 'datetime' date. So, we import 'datetime'. There's a 'century_start' variable that I'm setting up here, which sets up the date as, the first day of 2000, of course, century could be 2001, and there's a big debate about that. Technically, it's 2001. However, let's assume it's 2000. So, the century started on January 1st, 2000. They're the function in the 'datetime' library, which works on 'date' objects. So, this is telling us that, from the 'datetime' library, take the 'date' object, and apply the function 'today' to it, all right! So, we are disambiguating in here, from getting the library name, getting the 'datetime'—the...the date data type, and calling a function 'today', which is going to return today's date. And, we put inside this variable today. So, we get 'century_start' 'today' and this, and we can now do 'today' minus 'century_start'. And, we know that there are 600—'6,338 days' between January 1st, 2000 and 'today', which is the day I'm recording this, whatever day you're running this, of course, it will be different, okay!

The day I'm recording this is 2017-05-09. Notice that our time difference is '6338 days', zero hours, zero minutes and zero seconds, that's because 'timedelta'—and this is a 'timedelta' object, a...a duration—time duration object, contains hours, minutes and seconds as well. So, what we can do is, we could just extract the number of days from that. So, we get out 'today' minus 'century_start', which is this entire

thing here '6338 days, 0:00:00', and extract only the day's value from that. So, the day's attribute of that thing tells us that we are only 6338 days, for what it matters, formatting-wise, it's a little bit better.
Then, there's a 'datetime' object, which is very useful, because it gives us date and time. So, if you're looking at, for example, passenger flows on the New York City subway, and you're getting the time people swipe their Metro cards in, when they walk into the subway, then you—you're going to get a date and a time, down to the nanosecond. And, that's what we get in 'datetime' objects. So, we say—we start our 'century_start' again, but this time, we're just going to say '2000' and '1' '1' '1' '0' '0' '0' that means, it started at midnight of 2001, January 1st—2000, January 1st. And, we get 'time_now', and this time rather than using the 'today' function on a date data type, we're going to use the 'now' function on a 'datetime' data type.
So, 'datetime' is another data type. So here, we actually have the 'datetime' library, and the 'datetime' data type that's a part of that library. They're the same name, but they're two different things, right! That's the library and that's the data type. Then, we get the 'now', so this tells us what the time right now is, down to the nanosecond. And then, we print the '2' here, and then we find the difference, so we can now see that this tells us that we are '6338 days', 10 hours, 25 minutes, 06.840009 seconds into this century, as of right now. So, that's 'datetime'. So, that's—so, we have dates, we have datetimes. The nice thing about the 'datetime' library that it also checks the validity of whatever we are entering. So, if I want to create, for example, a date that says '(2015,2,29)', then I get a exception, which says, "Hey, you can't do that, that the day is out of range because in 2015, February, there were only 28 days." If I try '2016', however, which was a leap year, then that works fine. Not a problem, right!
And, the same thing for hours and minutes. If I try something like this, which says, 2015/2/28, 28th of February, 2015, the hour is 23, minutes are 60, then I get a...an error that says minute must be a value, or minute must be between zero and '59'. So, again, it checks all that, and it has a full calendar, so it will know what are leap years, what are not leap years, you know, the whole works.

Video 7 (14:11): Datetime library II

The next type we're looking at is 'timedelta' objects. They store the duration between two points of time, so we've already seen examples of that, and what we can do is–we can, given that we have the start of the century, here in, 'century_start', we've got the 'time_now', which is a 'datetime' object so it has date, as well as time. We can print the number of days, by just extracting that attribute. We can get the total seconds, this is going to be a little bit different from just doing seconds, because seconds will tell you the number of days, and the number of seconds for that day, the extra, that is over and above, like 600,338 days plus so many seconds, whereas, total seconds will tell us, the total number of seconds from the beginning of the century to now. So, that's the total number of seconds from the beginning of the century to now. If you want to convert it into minutes, then we can divide that by 60. If you want to convert it into hours, we can divide that by 60.
If you want to convert it into seconds—into days, we can check that, we already have days but we could divide that by 60 as well, and if you want to convert to years, it's a little bit complicated, because if you divide it by 365, then you're going to be messing up, in terms of leap years versus non-leap years, 365 days versus 366 days. So, you need a little bit of a complicated calculation, and there are other libraries,

that deal with this as well, but we're not going to worry about that, okay! So that's the—what we can do with 'timedeltas', and the fourth kind of object is the 'datetime'—'time' object. 'Time' object is a time of day, that is, disconnected from the date itself. So, let's take a look at this here. So, we have the 'now' gives us the date and time right now. And then, we can extract from this, the time part of it, the time part is going to be just the time without the data tabs. So this is '10:28:06.130552' This is so sort of useful if for example, you're looking at daily data and you want to find the– let's say, you want to bucket the times and we will do an example like that, so that you want to see what is the passenger flow between 8 a.m. and 9 a.m., between 9 a.m. and 10 a.m., etcetera. In that case, you don't care about the day as much, you're not... you're not concerned whether it's January first or January tenth, or whatever, you're just concerned about the time aspect, so you can get rid of the date completely and use the time. And then, that's...that's really when you use stuff like this. So, that's when—maybe bus schedules, if you were writing an application that contains bus or train schedules, then again you don't care so much about the day, you just care about the arrival and departure times.

So, those are our four datatypes, and we can do arithmetic operations on all that stuff here. So, for example, if we have, we can add to 'timedelta', to day or date, and get a date, or a date and time, to get a new date and time. So here, for example we've got the day, today, which is sitting in a variable called 'today', and we can add a 'datetime' 'timedelta' object, to this, which says, five days, so we're going to add five days to this. So, this tells us, here, that today, actually the day that I'm doing this is the 9th of May, let me just put that in there, so that's today. It's the 9th of May, so five days later is the 14th of May, and we've added that to that. So that's...that's good for, when you want to make calculations, on time inside your application or analysis. We can do the same thing with 'datetime' objects. So, here we have—'now' is 'datetime' dot 'today', 'today' is the same as 'now'. And then, we can add five minutes five seconds later to that, so we get this here.

So what we are saying is, you want to add five minutes and five seconds to our current time, which is this variable 'now', and then we get five minutes and five seconds later, which says— this is, of course, right now it's 10:35, let me add that back here. Now, so it's 10:30, and this will now become 10:35. So that's—we can do that. We can look at the difference, we've already seen that, so this another example. So, this is 'now' and we can get five minutes and five seconds earlier, and that tells us that five minutes five seconds earlier was, 10:25, it's 10:30 in the morning right now. However...however, you can't use, 'timedelta' on time objects, and if you do that you get what's called— you get a type error exception and there's a good reason for this, right, we'll come to that in a second. So, let's say we've got, we are looking at the 'time_now', and we are extracting only the time element of that right, so that's what we get here 'time_now', which is just the time element of our time. Then, we want to take the time, let me run this. And so that's the time, that is right now, right! And, we want to now, add thirty seconds to that, so I created a 'timedelta' object called, 'thirty_seconds', which is a 'timedelta' object and seconds equals thirty and I do, 'time_now' plus thirty seconds, and what I get is, a type error, it says you can't do that, you can't take a 'datetime' dot 'time' object, and add a 'timedelta' object to that.

There's a good reason for that, if you think about it, is it a bug or is it a feature. So, it's actually turns out to be, a feature, because if you think about it, let's say, you're flying from New York to let's say, to Yangon in Myanmar. Typically, you would leave New York at about, you know, eight in the evening and on day 1, let's say on the 10th of May, and you would arrive in Yangon in sometime in the morning of the

12th of May. So, if you have time objects, you would say I'm leaving at 9 p.m. and I'm arriving at 7 a.m. But you're actually arriving at 7 a.m., two days later, and that's not captured in just pure time objects. So therefore, within Python itself, the decision is maybe we'll not allow 'timedelta' calculations, on time objects, because there's this confusion as to whether you've crossed, more than a day or not, right! It's a...it could be a problem.

Therefore, for consistency and making sure you're always getting the right answer, we don't allow that. But of course, this is Python, so you can always, write a little function that does that, and you've got to get used to this, when you're doing data analysis, you're going to write lots of functions. So, we write a function called, 'add_to_time', that takes the time object, takes the 'time_delta' object, and what we do here is we just create an artificial date, so this is the year 500—the January 1st of the year 500, and we add that into our created 'datetime' object that takes that, and takes the hour, the minute, and the second from our 'time' object, and constructs a new 'datetime' object called, 'temp_datetime_object' And then, what we do is we add the 'time_delta', that is the time, sorry this one, the 'time_delta' to the new 'datetime' object, the 'temp_date_time_object', and return just the time component of that.

So, this is the new object that we are creating, and from that we're just returning the time component. And, that will pretty much do it for us, so we now get this of here, because keeping in mind that, we will have the same problem that if we, span more than a day, then our application is not going to capture that. We could force it to do that, if you wanted to do but we are not doing— our functions are not capturing that, we could force it to do that, but we are not doing it. Then, the next thing you want to do is that if we have 'datetime' objects, typically, you're going to get the date and time from some kind of input device, either a file or from a webpage or someone is going to enter it. So, whatever data that you get is going to come in the form of a string. Since, it's coming...it's coming in the form of a string, you need to convert it into a 'datetime' object to be able to reason, you know, to be able to reason with that time or date, or whatever.

So, since you're getting that as a string, we need a way of converting it, and there's a very nice library called, strip time or a function called 'strptime', s-t-r-p-time here, that does the conversion for us. So, if you go to this website over here, and let me just see if I can do that without messing anything up. Yeah, I can. This shows us that we have various formatting codes, for each type of data or time. So, for example, here it says if you are person 'b', then that's the month using the locale's month name, which may be abbreviated or full name. So, you might be able to do, something like, APR-01-01 for 1st April, sorry, 'Apr-01-2001' for 1st April 2001. So, what this is saying is, that you can use 'Apr' or 'April' if you're locale, that means your computer locale is defined as an English-speaking locale. If it's some other language, then it will be whatever the abbreviated or full form of the month is in that language. So, there are various formatting stuff, so we'll take a look at how to use these.

So, let's say we have a date here, that is, '01-Apr-03' right, so that is 1st April 2003. So, the...the codes that correspond to this are percent 'd', which is the code that says, essentially, the day of the month, with or without a leading '0'. So, either '1' dash 'Apr' would work or '0' '1' dash 'Apr' works. Percent 'b', which says, we just saw that, that's your code that says, you're going to use the name of the month and the language of the location where you are at or where your computer thinks you are at, whatever you're defining. If you're using a Spanish computer, for example, then it's going to use Spanish names. And then, '03', which is, percent 'y' which says, that you will use a two-digit year and or a four-digit year

actually, either is fine. And, if it's two digits, then it's going to use a code where, it says if the date is, I think if I'm not mistaken past 1966, let's take a look at that, percent 'y'. So, it says here the year within century, when the century is not otherwise specified, so in our case it's not specified, then the values in the range 66 to 99 shall be from 1969 to 1999 and anything from 00 to 68 will refer to the years 2000 to 2068, that's the way it does it.

So obviously, if you, once you hit 2068, we're going to have a problem. So we are just sort of pushing the problem into the future, if we do this, but often that's what you get. You have to work with the data you get, right, so often that's what you get. So, let's assume that's what we get here. So here, what we can do is, so we specify the date in a string format, so this is our date '01-Apr-03', in string format. And then, we give a formatting string after that. The formatting string tells us, what kind of format this variable is using to represent the date, the string is—the string you use to represent the date. And so, that has to correspond to what this is. And notice, we have dashes over here, and the dashes correspond to whatever was there, right. So, if there was a colon, then that should be a colon, if there's a slash it's got to be a slash or whatever right, so this is exact format. And this, we run this, and it tells us that we now get a 'date' object that is giving us this 'datetime' object, actually, because we're using the 'datetime' object here. It's telling us, it's '2003-04-01 00:00:00'. So, that's strip time and we'll take an example for [inaudible] that will do a little bit more with that.

So, unfortunately nothing there for 'timedelta', so you can't take 'timedelta's and convert them into, in the form of strings and convert them into a 'timedelta' object. So what you would need to do is, you would need to write your own function for it, or do the conversion, explicitly. In this case, what we do is, we say all right our string is, hours colon minutes colon seconds. Therefore, we're going to extract the three elements, the hours, minutes and seconds using the split function, split on colon over here, right, we can see that splitting on colon there. And then, construct a 'timedelta' object by saying the hours are the hours, the minutes are the minutes, and the seconds are the seconds.

Notice that we use the unpacking assignment here to extract each element that was extracted from the splitting function. And then, we can functionalize that if we want to, but you can take a look at that on your own. And the reverse operation, like if you have a...a 'datetime' object and you want to write it to a file or send it across the Internet or do anything, outside your application with that object, then what you need to do is, you need to convert it into a string because data flows in the form of a string. So, the reverse operation is called 'strftime', so we have 'strptime', which does the conversion from a string to a 'datetime' or 'date' object and we have 'strftime', which does the conversion from 'datetime' or 'date' object into a string, which you can then use to send to something else.

So here, for example we're getting our date now, which is in the form of this format here, or rather this format. And we are saying, convert this into a string with, percent 'm', percent 'd', slash percent 'd', slash percent 'y', so we get percent 'm', slash percent 'd' slash percent 'y', and then a space, and that's the space and then hours, minutes, seconds, percent 'h', percent 'm', percent 's', and that comes here, with colons in the middle and notice we have the colon, so the colon comes here. If we don't have a colon, it's not going to come there right. So just to, you know, clarify what this stuff is. If I change this to... to this, for whatever reason, right, and I run this, then I'm actually going to see the 'yy' over there, right? So the formatting is actually just taking exactly, what we're giving it, and replacing it, by you know putting

that in the string, that we are using in the conversion, from a 'datetime' to a string object. So that's the basics of the date time library.

Video 8 (11:15): Bucketing time Part I

So, let's end our Python by unit by looking at a nice example, an example of bucketing time. In New York City, there's a number that you can dial, '311', that is a complaints hotline that you can make complaints about all kinds of things, potholes or bus delays, or a taxi complaint, or heat, or noise, or whatever you want. And these complaints are all recorded and stored, and New York City has a very nice data delivery service, where you can get data almost everything— any kind of data that New York City produces is made available to the public. So I've got some of that data here, and I've modified it a little bit, and I put up a little brief sample file, that you can use and we'll take a look at how that works. Okay! So in this sample file, what we're going to do is, we're going to, take our— the date of '01/01/2016', and compute the average processing time, for each hour of that day. So, what we're going to see is, what was the average time it took to complete processing a complaint that came in between, let's see midnight and 1 a.m. of January 1st 2016. So, these are—they take a couple of days, they take a few days to complete. So, the result we're going to get is a number of days, but we are looking at the time, that the complaint came in, and how long it took to complete it, just as an example, And we want to see, our goal really is to see, is there a time of date difference. Are you more likely to get your complaint resolved faster, if you complain in the middle of the night, or is it going to be faster, if you complain in the middle of the day or- are there some differences, in time, across the day. So, what we're going to do is, we're going to bucket our day into hourly buckets, So, you get a bucket from zero, midnight to 1 a.m., from 1 a.m. to 2 a.m., 2 a.m. to 3 a.m., and for each bucket, we're going to calculate, the total processing time for all the complaints, calculate the number of complaints that were received, and divide those two, the total by the number, and get an average processing time for each bucket. That's our goal here. So let's take a look at the data, for starters, and this is an extract really- the data is very comprehensive. You can go and look at it, Google it, and see the data. There's lots of stuff there.
I've calculated the processing times, and I've extracted the time that it takes— the time that it arrives. And we are using this function called, 'head', which is a UNIX function, and will not work on your Windows machine, but it will work on your Mac. The exclamation point here essentially says, run the command from your system, not from Python, so we're running this from system and it's saying, get me the first few lines of the file, 'sample data.csv', that's in your—that file is in your—online, you can download it, and take a look at it. So, what does this contain? This contains a 'datetime' string that tells us here, for example that this is '2016-01-01', and the time is '00:00:09'. It came it at 9 seconds of the new year of 2016. And the time that it took to process it was point zero eight one days, okay, these times are in number of days, that complaint.
So if you look at that site, you're going to see that, there's a lot of detail about the complaint itself, but we are ignoring all that, we're just looking at the actual processing time. It's just a quick example really here. So when you're doing something like this, the first thing you want to do, really is take a look at the data itself. So that's what we've done here. We said, okay, let's take a look at the data, and see what it looks like, so we know now what it looks like, so let's read it, into our program. And then, we haven't

looked at this, but in Python if you want to read a file in, you can use the 'open' command, it says, open 'sample_data.csv', and we want to open it for reading,
The 'r', over here, says that, we're going to read it. And we're going to put the file handle in a variable called 'f', so 'f' is going to refer to the file that we have opened, and what you can do is, you can use this, 'with' block. So what the 'with' block does is, it sets up a block and says, that this 'f' is valid only inside this block, which means, that we don't have to explicitly close the file, which is kind of nice. At the end of the block, it will automatically get closed, and we'll be done with it. So that's what we're doing here, so, we say, with open 'sample_data.csv', 'r', so we're opening 'sample_data' for reading. Then we have our file handle, 'f', and we're going to now, loop through, the file handle 'f' for every line in 'f'. So it's going to assume that a line ends with a 'backslash' 'n', an end of line character. And we're going to take each line, in turn, and, we're going to create a list called, 'data_tuples', and 'data_tuples', are going to be tuples of the time, and the processing time.
So we're going to append to this, we'll strip the line, strip to get rid of any backlash ends, or trailing, leading spaces, whatever, you know, mucky stuff. And then split it on a comma, and, so what we do is, we get two elements, and this will give us our 'data_tuples', which is going to look something like this. So we get a data list really, not tuples. I should call it, list. The reason I can't make it a tuple is, because I'm going to modify it. So remember, tuples are immutable, I can't mutate them. So here, I could've used tuples here, but I want to modify the tuple later. Why do I need to modify it? Well, I need to convert this into a 'datetime' object, I need to convert this into a floating-point number. So I need to modify the tuples. So that's what I'm going to do here, so I get this stuff here.
So now I'm looking at this, and what I want to do is, of course, convert the first element into— I should make that a list. Just correct that, list. So I want to convert the first element into a 'datetime' and a second element into a floating-point number. So that should be, again, pretty straightforward. All I need to do is, I need to go to the file, the page we looked at earlier, the...this page, and figure out what the right format is, right. So I leave you guys to figure that out, and over here, we—you need to do the reading yourself, but take my word for it, that percent 'y' here indicates the year, in four digit format. The dash, is of course the dash. Percent 'm', is the month in number format, with or without leading zeros. Percent 'd', the dash is a dash again, Percent 'd', is the day in numerical format, 01, 02, 03, or 1, 2, 3, with or without, the leading zeros.
There's the space, so we have the space here, then we have hours, the minutes and the seconds with colons in the middle. So that's our format string, and we can test this out, and see if it works. And it doesn't work because we don't have the 'datetime' library imported, right. So, 'import datetime' and run that, and we get, yes, it works for us. So, we get 'daytime' dot 'daytime' object with '2016, 1, 1, 0, 0, 9', because remember that was— this is the date that we are converting, right. We've taken the first date and we're just using it as an example. So we know our format string is correct, so now what we can do is, we can iterate through our list, of lists, that contain all these elements in it. And for each element, we'll take the first element, convert it into a 'datetime' object using the format string, and take the second item and convert it into a float, by just calling the float function. So I've taken the whole thing here again, so this is, we've already done that, 'import datetime', for 'i' in range zero to length data tuples, we take the first element and convert it into a 'datetime' object, and take the second element, that is the oneth element, and 'con—float' using the float function, we convert it into a floating-point

numbers. And take a look at this, and yes, now we have 'datetime' objects and floating-point numbers, in our list over there.
So that's great, we've got that. Now we can figure out the hourly buckets. So this should be pretty straightforward, if you think about it, because we have here, a 'datetime' object, and the 'datetime' object contains the year, the month, the day, the hour, the minute, the second. What we want is the buckets for zeros, ones, twos, threes, whatever, right! So all we need to do is, we take our— any one of these things here. So I'm going to take an example, the very first element that we have, and pull out the 'datetime' object value from it, that is the first element in that list, and ask for the value of the hour attribute, And if you see that, it tells us, the hour is zero, right! Which is, what it is, so that worked. So what we can do now is, we can run this on our entire set up here. And I want to use a Python technique called, 'list comprehension', which is a...a very useful technique. So what that does is, it in a single line, it really runs an entire loop for us. So what we're doing is, we're saying, we want to construct this list. So say, we're making a list, so we put square brackets, and the list is going consist of elements, and each element, is going to be a tuple. And the tuple is going to contain the hour and the minute—sorry, the hour and the processing time for every 'x' so 'x' '0', the first element, and 'x' '1', the second element, for every 'x' that is in data tuples. So this is, if you think about it, is actually equivalent to, thing for thing or rather 'x' in 'data_tuples'. We want to replace the data tuple by— I'm not going to do that, I'll just print it. Print 'x' '0' dot, hour, comma, 'x' '1', and okay, run that. I get all these things there, right! So that's 'x' '0' and 'x' '1', right! So, that's pretty much what I wanted out of it. So this is a very handy way of creating lists, or creating dictionaries or anything like that, in this case, it's probably not helping us a whole lot, but it's still—you know, in a single line you can express it. And we can—if you make this more and more complicated, then, everything comes in a single line, it makes your program very compact. So this is a very nice thing in Python that is well worth looking at and you can take a look at that. So here, what we do is we construct the same thing, and we do our 'data_tuples' there, and we have now an hour, minute here. And I should probably take a look at that. And that's what we've got here right now. Okay, so we have our bucket, so the first element in our list over here, in each tuple in this list, is the bucket number, and the second element is the processing time.

Video 9 (09:20): Bucketing Time Part II

Let's put it all into a nice little function that does that so, here what I'm doing is, we got a function called, 'get_data' that does all this stuff that we did before. We come to the format string. And now, instead of doing, creating my 'data_tuples', in two steps, that is, one step where, I do the conversion from, the string to a 'datetime' object, and the second step, where I do the conversion, from the string to the floating point object, and then...and then, take that and then, you know, extract the hour, what I want to do is, I want to extract the hour, in a single step over here, and do the floating point over here, and construct everything in one shot. So this, I write this function here, and one shot—in one single list comprehension statement, so I write that there, and I can test that out, And, yeah, that works, right! So great! So now, what I need to do next is, I need to actually do the counting, so what I want to do, remember, is compute the averages by bucket. So what I want to do is, calculate the total processing time for all elements in a bucket, and divide by the number of elements in that bucket. So, to do that,

what I'll do is, I'll create a dictionary over here, the dictionary is going to have the bucket number as a key,

So the keys are going to be zero, one, two, three, all the way to 23 for the 24—zero to 23 hours. And then, for every item in this list, that means for each tuple that contains a bucket and a...a value, the processing time value, I'm going to check, if I already have that hour in the—in my buckets. So, buckets initially is an empty dictionary, and so, what I want to do is, let's say, I come...I come in with a new data item. So, the first time when I see bucket number zero— so when the first time I see bucket number zero, we are not going to have an element with a key zero in the dictionary. So, the first time I see it, I add it in, I say, 'bucket' 'item' '0' is the count, '1'— this is the first time I'm seeing it, so the count is '1'. And, the total sum of all the elements, so far, that I've seen, is the processing time for that item. So, 'item' '1'. So, the first time I see it, I just add this in. So, let's say, for example, I see this first, so at this point, I haven't seen anything else, so maybe, I should take the very first one anyway. So that's the very first one, right! So, at this point I haven't seen anything for bucket zero. So, therefore, at this point I know, that this is the first one that I have, so I have one item, that is this, and then, the total of items that I've seen so far is, processing time is 0.0815.

So that becomes this, 'item' '1'. That's...that's what I get the first time I see it. The next time, when I see a...a new item for bucket zero, I already have, an element in my dictionary with key zero. So, what I can do then is, if 'item' '0' is in buckets, that means, it is inside my dictionary, then, I add one to the count, and that's that. And, add the value of processing time to the total processing time, and that's that. So, at the end of all this, I get my 24 buckets, from zero to 23 with a count of elements, and the total value of the processing time, and all I need to do now is to divide this number by this number for every bucket, right! And I get my—the result of what I want to do. So, that's what I do here, I print the hour, and that's the key for, every key-value pair in 'buckets' dot 'items', remember, that pulls out, the key as well as the value. So, for each key-value pair, I print the key, and I print the value one divided by value zero, value one, is this, and value zero, is that. And I can see this, and this, is what I get. So looking at this, I mean, this is just one day. So, bear in mind this is one day. I can see, yes, there is some kind of effect, where in the middle of the day there is, over here, we find that the processing times are higher, and early on, and late evening, they're not that bad, evening is pretty high. But, you know, after midnight, of course, this is January 1st.

So hopefully, everybody was, this should have actually be lots of complaints, but there might have been just like noise, or people going crazy after partying, or whatever happens in January 1st; right? So, we got that. So now, what we want to do is, we can do this for one day. But, obviously, we want to do more than one day. So let's take all that we've done so far, and put it inside a nice little function that's going to get us our bucket averages for any file that we can give it, right! So, that is one day. And, if you want more data than one day, then we can just give it a new file. And, we get our results for the new file. So, what does this do here? We define this function, get, this is, you know, generally, when you're working with data, you want to be doing a lot of this stuff. Take all the stuff that you've done, work line by line, collect it all, and put it into a nice little function. And then, you can then call as many times as you like, or put it inside other functions, and, you know, stuff like that.

So, here we got a function, 'get_hour_bucket_averages'. And, inside that the first thing we do is we define a function that returns our tuples that contain the hour and the processing time, these tuples.

Here is a list containing these tuples, right! So, that we've already done. And, we will—we have that function 'get_data'. I've just copied it down and put it on over here. And, that's what this is, right! So, we've got that. Then, we want to actually do the bucketing, so we create a dictionary of buckets, with empty dictionary. And then, we say for everything in, and we call our function 'get_data' file name, that's this function, that we defined over here. So, this is going to return that list of tuples for us. And, we do the bucketing, exactly the way we said before. And now, instead of printing stuff, we're going to actually return the key, and the—so this is now another list comprehension that we're doing. We're returning a list, that contains each key-value pair, and the key, in this case is, the key is the bucket number, and the value in the tuple, that we're returning is, the total processing time divided by the total number of cases or complaints in that bucket, for every key-value pair in our bucket dictionary.
So, every key-value pair in our bucket dictionary, we do the calculation, and we return that list. So, we can do this and we can take a look at 'sample_data' and we get our, the same thing again, 0.65, 2.96, and, yes, 0.65, 2.96. So, just for the heck of it, I took a larger file, which I'm not including because it's a much larger file there, which contains all the data from January to September 2016, and said, okay, let's see what that can do for us. And, the data itself online is from 2010 to the most recent month, which is, in my case, is end of April 2017. So, you have a lot of data that you can work with. The data is in a huge file, so if you want to just get a couple of columns, you need to do some pre-processing, but you can actually do that, and extract the two columns that we have, and then, run this for all the data itself. So, I'm not going to run this, because it takes about a minute to run. And, you know, you don't want to be sitting and looking at me for a whole minute, so, with nothing happening, so I ran this before we had the class.
So, if you look at this stuff over here, we find that, of course, now we're looking at a much larger data set, and not just one day, so we don't find the same variation that we had before, but we do see that, there is some kind of a middle-of-day effect, Because earlier on in the day there is 4.4, 2.8, 2.8, 3.5, 4.4, etcetera. Then, as we get to nine o'clock, ten o'clock '15', '16', '17', 5 p.m., 6 p.m. This is slightly higher than earlier in the day, and then, it drops again later in the evening down to the you know fours and threes in the number of days. Remember, this is the average number of days, it takes to process a complaint. So, I don't know, you know, I'm just making this up, as we go along, but it's quite possible that, the kind of complaints that come during the middle of the day, are actually harder to resolve, than the kind of complaints that come during, at the beginning of the day or at the end of the day, right! Either early in the morning or very late at night, so it's quite possible, and ideally, what we would like to do then is, because the data contains lots of information on what the exact complaint was, which city agency was involved, which borough it was in, even the address, even the location, the latitude and longitude of where, the complaint was—occurred, like, if there was a pothole, where the pothole was, the taxi complaint, where the taxi was, was a heating complaint, where the unit, the house is, the apartment is, that kind of stuff. So, we can actually take that data and figure out, why we are getting this pattern, assuming that, this pattern actually carries through from 2010 to 2017, because, again, note that all we are looking at, is a small slice of the data, which is nine months in 2016. So, there could be other factors that affect that. But just to show you that this is what really happens with data analysis, and as we go further into our Python-based data analytics stuff, that we are going to be doing for the rest of this course, we will look at these questions in more detail.