Week 3
Video Transcripts

Video 1 (03:52): Python data types

Hello, everyone! So, let's start with the basic data types in Python. And, just like in any other language, you have numbers, and the numbers are either integers or floating point. And, the way we specify them in Python is that they are determined by the context in which they appear. So, here for example, we have the number '47'. And, the number 47 is an integer. Therefore, the variable 'number_of_students' takes an integer value and it automatically becomes an integer variable, which means that all integer operations are now applicable to the variable number of students. Similarly, we have '93.74', which is a floating point number, a decimal number. And, the variable 'purchase_price' takes a floating point value, and it becomes–any floating point operation is now applicable to 'purchase_price'. You can do numerous operations with numbers. So, let's quickly walk through these. And, we'll see that for every session that we have, we have our Python notebook. So, it's a good idea to get used to using that. And, in iPython Notebook we have cells. So, the cells here contain pieces of code.
You can call them code fragments and we can... we can execute them and see the...see the result. Whatever the code fragment produces is then displayed in an outbox under the inbox. So, here for example, we see that we've got an 'In' one, which is our '21' times '21', and an 'Out' one, and that's the value returned by that operation, note that it's not a print statement.I It's just a value returned by the operation. The anaconda Python, iPython Notebook is a very useful way of building programs. Obviously, you're not going to deliver that program to someone, but for building it, it's actually a very neat way to do that, and it's also good way to present results of data analytics. But anyway, let's get back to our operations with numbers. So, we–there are a whole bunch of them here. You can multiply stuff, you can– if you multiply two integers the result is an integer. If you multiply an integer by a floating point number, in other words, if either operand or both operands are floating point numbers, then the result is a floating point number. You can divide one integer by another. In Python three, it's always going to return a float. In Python two, it's different, but Python three is always going to be a floating point number. You can always return an...an integer by using two slashes, so two slashes is integer division.
However, note that if either of the operands in a slash slash operator are float, then the result is also a float. And, there's some tricky stuff like, for example, if you divide '21' to a integer division by '3.2', we're going to get the lower integer bound of '21' divided by '3.2', which is '6.0' and then return– it's a ’6’, which it will...it will report it as a floating point number. Similarly, we can find remainders, and a remainder, if you have both integers, is also an integer, '21' divided by '4' gives a remainder of '1'. If you divide it by–if you find the remainder of '21' divided by '4.4', which is a floating point number of '4.4', then what you really get is '21' minus '21' slash slash '4.4', which is ‘3.4’. Try it and you'll see that. You get a–if you're raising the power, you use two stars, and of course a floating point raised to power is

going to return a floating point number. So, that's pretty much it. It's...it's what you would expect to see with numbers, so there's nothing very exciting about that.

Video 2 (10:57): Basic data types in Python

In Python, we also have strings. So, for example, we have a string 'x', a string variable 'x', and the value of the variable we assign, the value assigned to it is 'John'. So, what is 'John'? 'John' is a string. In Python, you can use double quotes or single quotes, and they're interchangeable. There's no notion of a character as being distinct from a string in Python. Let's take a look at some string operations. So, let's say we have a string here, which says– which is, 'Always take a banana to a party!' The term for a string is also a literal. Actually, for anything that is a value, is a literal. So, this is string literal, and 'x' is the variable associated with that. And, the key thing to note about strings is that they are ordered collections.

So, that's an important characteristic, because if something is ordered, you know, that like if you have let's say, '2', '5', and '7'. These are ordered numbers, and we know that '2' comes before '5' in the– on the number line, right, so let's say that's a number line, and let's do '5' and '7' on it. We know that '2' comes before '5' and '5' comes before '7', right, so, there's an ordering defined on it. So, when you have an ordered collection, what happens is that you can think of your collection as being inside a bunch of boxes. Each box is labeled '0', '1', '2', '3', '4', '5' etcetera. And inside each box, you have a value. So, we have 'h', 'e', 'l', 'l', 'o'. So, 'h' comes before 'e', 'e' comes before 'l', 'l' comes before the other 'l', and this 'l' comes before this 'o', right. And so, what we can then do is we can say– we can ask questions like, 'What is in box '0'?' And, that's what this is saying, 'x' '0' is saying what is in box '0'.

So, in box '0', we have the first character, which is 'A', and that's what we see here. The value of 'y' is uppercase 'A'. Similarly, 'x' '3' gives us the value in box three, which is the fourth character, '0', '1', '2, '3', and that's a lowercase 'a'. So, this is called the–this– the 'x' or some variable followed by square bracket, then this is an indexing operator. What it's doing is, it's returning– it's indexing the collection, and giving you the value associated with that location. So, in Python, that's pretty straightforward. In Python, you can also go backwards in a string. So, if you do a minus one, for example, what that does is, it takes you to the last character in the string, in this case, that is the exclamation point. You can't go beyond the end of the string. So, our string for–let's take a look at this string here. So, our string here is, 'Always take a banana to a party!', and we saw 'x' three will return lowercase 'a', 'x' negative '1' will return the last character, which is an exclamation point, and 'x' negative '2' will return the second last character. So, you can start going backwards with negative '1', negative '2' etcetera.

But, you can't go beyond the end of the string because there's no character there. So essentially, we are saying we've run out of boxes, so a good idea is to make sure that you never go beyond the length of the string. You can slice a string to get sub-strings out of it. So for example, if I do 'x' '7' comma '11' in our string over here, then what we are doing is, we are saying go to character number–character and box seven. So that's '0', '1', '2', '3', '4', '5', '6', '7'. And then, go all the way to '11' and extract that sub-string. So that's '7', '8', '9', '10', '11'. However, we don't actually extract the box– the character in the 11th box. So, we extract '7', '8', '9', and '10' and we get the value 'take'. That's what we get out of there. So...so, when you think of sub– of...of slicing, what we think is we have 'x', and we can do start

location and an end location, separated by a colon, and that tells us where we want– what kind of sub-string we want. We can also include a step– a third argument, a step, and that tells you how many characters you want to skip between each character that you draw.

So, for example, if I do–start with '0' and go to the end, note that if either 's' or 'e' is missing, then 's' assumes that it's going to be the beginning of the string or the most reasonable starting point, and 'e' assumes it's going to be the end of the string or the most reasonable ending point. And, we'll see what that means in a second. So, if you go here, we are saying start from '0', go all the way to the end of the string and go two characters... characters at a time, so what we get is a sub-string that has every second character in it. So, let's take a look at that and– so here this returns us 'take', that will return everything starting with the 't' in 'take' and going to the end of the string, because we haven't specified explicitly, an ending character. In this, we are saying we want to start with zero in the...in our next box. Start with zero, go all the way to the end, and take every second character. So we get this 'Awy' blah blah kind of stuff. If we skip the first two locations, the first two parameters, so we don't have a starting location, we don't have an ending location, and we want to go negative one, which means a step is negative one, then what the Python will assume is that the most reasonable starting point is the end of the string, the most reasonable ending point is the beginning of the string because we are going backwards, and then draw every character backwards, and we get a string, essentially, reversed. You can also search for sub-strings, and what this does is if you use the 'find' function, the 'find' function is going to return the location of the first sub-string that it finds corresponding to whatever argument you've given it.

So here the argument is 'to', it's going to return the location of the first 'to' inside the string. So, for example, if I look at our thing here, if I want to find 'to', I run this and I get that the 'to' is in location number 21. If, for example, I want to look for the lowercase 'a' and I do-- 'x' dot find lowercase 'a'. Oops! It tells me 'a' is at 3. Obviously, there are far more threes than– sorry, far more a's than there are– then just one in the string. What it does is, it returns the first instance that it finds in the string. If on the other hand, it cannot find the sub-string, then it returns a negative one. So, when you're looking for a string, you can always search for the string, and then if you get a negative one, you know it doesn't exist inside the-that sub–string doesn't exist inside the string.

The last thing about strings that are interesting really is– well not the last but one of the last things, is that they are immutable. What does immutable mean?

In Python, immutable means that you can't change its value, and if you try to change the value–so we have our 'x', 'Always take a banana to a party', exclamation point, and we trying to replace the character in location five by an uppercase 'C', you're going to get a type error, an exception, that's a type error because you can't change a string in place. So, let's take a look at the– what happens when we assign values to a string. So we assign the value 'Hello' to 'x', and then we say there's another variable called 'y' and it has the same value as 'x'. When we say it has the same value, what we're really saying is that they both point to the same location. And so, what we get–end up getting is 'y' and 'x' both are at this location over here. The function 'id' that we see in this print statement over there, gives us the location of a variable or any object inside Python. It tells us where it is in memory. So, this is in the same location. And, that's sort of an important concept because we know that now 'x' and 'y' point to the same thing. But, unlike in other languages, where if we replace 'x', then 'y' would also change because 'y' would be considered- like in C++ or C, considered an alias of 'x'.

In Python, 'y' and 'x' are just names and they're pointing to a location, and if we replace the value of 'x' and we give it, let's say, 'Always take a banana to a party!', then what happens is that 'x' and 'y' now point to different locations. And, we can see that here that they both are pointing to different locations, 'x' to this location and 'y' to this location. The next thing we can do with strings is we can concatenate them, and concatenation of strings really means we just add them up together. And, when we add them up together, if we have a string 'abc' and a string 'def', then this is going to give us a string 'abcdef'. In other words, 'def' will be just concatenated to the end of 'abc'. But note that these are all different strings. So, this is one string, that's another string, and that's a third string. So, if we want to use this, we have to actually change it, and...and make sure that we're assigning it to a variable. So here for example we are saying, we have 'x' is this, and then we have 'y' is this, and 'z' equals 'x plus y', and we print it we're going to get, 'Always take a banana to a party! Never forget', right, That's what we're going to get. So let's see how this works out in practice. We have a string 'x', 'Hello', and a string 'y', 'Dolly', and we add 'x' plus 'y'. So, we add this up, we get, 'Hello Dolly'. That's 'x'. That's 'y'. That's 'z'. That's the location of 'x', the location of 'y' and both they're different locations, and location of 'z'. They're all in three different locations and three different strings. It's important to see that here that we have to add the space. It's pretty obvious I guess, but let's check that out. So, we've got 'HelloDolly' over here with no space at all. Python not going to understand that we want to put a space, so what we need to do is add the space explicitly and put this– create 'z' with this additional space added into it.

Video 3 (01:44): Variables and values

So, it's now–it's worthwhile looking at variables and values just to get a sense for how Python deals with that. So, I'm just going to walkthrough this stuff here. We have, in this example, a...a variable 'student1' that has a value '87' and we see that in a certain location. That's the location where it is. We have a variable 'student2' with a value '95', and it's in a different location. If I equate 'student1' and give it the same value as 'student2', then if I look at the location of 'student1', we find now that it's at the same location as 'student2'. They're both in the same part of memory, the same location in memory, because that's what I've done. So, in Python, what happens is that you can think of values, explicit values that are non-changeable as sitting in memory permanently, kind of, you know. So, if you have the number '5' it's always going to be here.
So, if I say 'x' equals '5', then that's going to point here. If I say 'y' equals '5', then that's also going to point here. If I say 'z' equals 'x', then that's also going to point here. Then, in if I say 'z' equals 'z' plus '1', then what happens is not that the–what's...what's in– it's not that the whatever is in–the the location of 'z' changes, but 'z' just points to a different location. So, let's say there's a '6' here and 'z' now will point to this location over there. That's how Python works, right. And, the idea being that these numbers '5', '6' etcetera are immutable. They will never change. So therefore, they have independent existence and anything that points to them will point to them, and if you change the thing that points to them that will point to something different. Okay, that's the idea there.

Video 4 (08:47): Boolean types in Python

So, let's go on and look at Boolean data types. The Boolean data types are data types that evaluate to either 'True' or 'False'. And, in Python, just has a matter of syntax, false is always uppercase 'F' with a lowercase 'a' 'l' 's' 'e', and true is always uppercase 'T' with a lowercase 'r' 'u' 'e'. So, here we have an example, where we have 'x' equals '4', 'y' equals '2', and then we check whether the 'z' equals 'x' 'equals equals' 'y'. So, this 'equals equals' is a relational operator. It'll return 'True', if 'x' and 'y' are the same. It'll return 'False', if they're not the same. In this case, they're not the same so 'z' becomes 'False'. And, in this case, 'x' and 'x' are the same, so 'z' becomes 'True'.

Okay! That's the idea. And, we can always give values to those things. So, as a summary of all the operators that you can use in Python, for doing logical and relational operations, you can think of them being divided into two groups. There's the relational group, which checks to see- which compares two values, and returns 'True' or 'False', depending upon what the comparison- whether...whether the comparison asks for, what's 'True' or 'False'. So, 'x less than y', 'x greater than y', 'x less than equal to y', 'x greater than equal to y', and these will always return either 'True' or 'False'. So, when you use the relational operator, you're guaranteed either a 'True' or a 'False' or an error, if you haven't done it properly.

On the other hand, we have these operators, and these are logical operators. Logical operators are used to modify an existing truth value or to combine one or more truth values. So, the easiest one, of course, is 'not x'. What 'not x' will do is, it'll flip the value. If 'x' is 'True', 'not x' is 'False'. If 'x' is 'False', 'not x' is 'True'. So, 'not x' is always going to return, either 'True' or 'False', just like the relational operators. We also have 'x and y' and 'x or y'. What the 'and' operator does is, it takes two operands and it will return 'True' or it will take the value 'True', if both 'x' as well as 'y' are true, and it'd be 'False' otherwise. So, if either 'x' or 'y' evaluate to 'False', then this is going to be 'False'. That's the way to look at it. On the other hand, 'or', what 'or' does is, it takes the value 'True', if either of 'x' or 'y' are 'True', which means that it'll evaluate to 'False', if both are 'False', and every other case, it evaluates to 'True'.

So, let's take a look at this thing a little bit because Python has its own way of dealing, like with everything else, dealing with Booleans. In general, everything in Python has-you can think of it as having a truth value associated with it. So, if you take a-an number, say 'x' equals '8', that's 'True', because it's not zero. So, anything that evaluates to '0', or nothing is 'False'. So, a number that had the value '0' is 'False'. A string that has nothing inside it, an empty string, is 'False'. And, every other number or every other string is automatically 'True', okay! So, that's the way to look at it in Python. So, what this does is, it makes the idea of expressions a little bit funky, in the sense, that you may be expecting a logical expression to always return 'True' or 'False', but it doesn't have to. What it does is, it returns whatever value makes sense in that context. So, if we have here, let's take a few examples, so let's say, we take the number '8', 'x' equals '8' and we look at the value of 'x', that's '8' and the value- the Boolean associated with that is 'True' because '8' is non-zero. If we take the empty string, then the value of the string is of course an empty string, and we see that here by a little empty space there, which you can- next to the 'False'.

You can't really see it, but it's there. And, the value of that is 'False' because it's an empty string. Then, we look at logical operators. When we look at logical operators, we look at the 'and' operator. So if we have 'x' is '4' and 'y' is '7', 'y' is '5', sorry, and we want to see what the value of 'x greater than 2' 'and' 'y greater than 2' is. We get a 'True' because both 'x' is greater than '2' as well as 'y' is greater than '2' are

both 'True'. If we look at 'x greater than 2' 'and' 'y less than 2', we get a 'False' because they're...they're not- it's not true that both of them are 'True'. One of them is 'False'. If any one is 'False', then automatically an 'and' returns a 'False'. If 'x is less than 2' 'and' 'y is less than 2', in this case both are 'False', therefore that's also 'False'. On the other hand with 'or', we are looking at one of them being 'True' to get a 'True' value, and we'll return 'False', only if both are 'False'.
So, if 'x is greater than 2' 'or' 'y is greater than 2', that's going to be 'True'. 'x less than 2' 'or' 'y greater than 2' is going to be 'True', because though 'x less than 2' is 'False' 'y greater than 2' is 'True', so we get that. On the other hand, if we have 'x less than 2' 'or' 'y less than 2' both are 'False', therefore the result is also 'False'. If we take a 'not' of any of these things, so 'x greater than 2' 'or' 'y greater than 2' is 'True', therefore the 'not' of that is 'False'. And finally, we look at this stuff here. Let's take a look at this. So, we have print 'x or y'. So, 'x' we know evaluates to 'True', because it's not zero. 'y' we know evaluates to 'True', because it's not zero. However, the result of 'x or y' is going to be the value of the last expression evaluated, and what...what Python does is, it is opportunistic in the way it evaluates these things.
So, when it's looking at 'x or y', it tells itself, "Hey! Wait a second, that's an 'or'. So, if 'x' is 'True', I don't really care whether 'y' is 'True' or not. Therefore, I'm only going to evaluate 'x'." It's not going to look at 'y' at all. Since 'x' was '4', what we get here is the value '4'. It hasn't even looked at what the value of 'y' is. We...we don't care. The value of 'y', just for reference, was '5', right! It...it ignored that completely. In the other hand, if we look at 'x' equals '0 plus 3', then this case 'x' is 'True'. because 'x' is '4', which is not zero. Therefore, it has a Boolean value of 'True' associated with it. However, Python says, "Hey! Wait a second. I know 'x' is True, but I don't know if 'x' and whatever comes after it is True, without actually evaluating whatever comes after it." So, it does that evaluation as well, and it returns the value of the expression that is the second operand in this 'x and y', or something and something expression. So, here we get a '3', which is '0 plus 3'. So, notice that 'or' and 'and' are the two operators here. These two-that may not return 'True' or 'False'. So, may or may not, 'True' or 'False'. It all depends on what the last expression evaluated was. And, that may not always be very clear in the sense that if, for example, I'm looking at something like this. 'True and 24', all right!
So, what is this going to return? So, here we have a 'True' on one side and there's an 'and' and then there's a '24'. So, what Python's going to do is, it's going to look at these as independent expressions, right! So, 'True' is one expression and '24' is a second expression. And, what it's going to say is, "Hey! There's an 'and' in the middle, so I need to check, if both of them are 'True'." So, in this case, it's going to return, it's going to say, "That's 'True', great! So, let me check '24'. That's also true." So, it's going to return '24'. However, if I change this to 'False and 24', then what happens? So now, what we see is that Python is going to look at this and say, "Hey! Wait a sec. This is false. So, I don't really care what's on the other side. I've already found something in an 'and' that is 'False', therefore the entire thing is 'False', and I'm not going to bother with what's on the right-hand side, what's the second operand." And, it returns a 'False'. So, we actually get different data types returned by this expression, depending upon whether it evaluated the first operand only, or the first and second operands. So, that's something to note and it's a tricky kind of factor.

Video 5 (05:25): Assignment Operations

So, let's talk about variables and assignments. Generally, just as a sanity check here really– we all know this, a variable must be declared before you can use it. That's almost a given. So, you can't do something like... like, right, your program can't say 'x' equals 'y' plus '5', if 'y' hasn't been declared before, right! That's...that's the bottom line. All variables are...are almost in all–every case, not everywhere, but in most cases–the way you will give a initial value to a variable is by putting it on the left–hand side of an assignment statement, okay. So, that's an assignment statement. The equal to sign indicates an assignment statement. And, you can think of an assignment statement as having two parts. On the left-hand side, there's a name or identifier, and on the right-hand side, there is an expression. What Python is going to do is, it's going to evaluate the expression and whatever value the expression returns is going to be assigned to the variable name, okay, whatever the variable name on the left-hand side is. So, it follows, as a corollary, that the expression has to be evaluatable, which means that every variable that is included in the expression must already have a value, otherwise you can't evaluate it.

That's a pretty straightforward. So, in general, you can say that the left-hand side of an assignment statement is almost always a single variable name, and the right-hand side must be resolvable to a value. If you can follow this rule, you're in pretty good shape in Python. There are four kinds of assignment statements in Python. There's a simple assignment statement, 'x' equals '5', which is, of course, the kind we've seen so far, a name on the left-hand side and an expression on the right-hand side. You can do multiple assignment, 'x' equals 'y' equals '5'. In this case, what happens is that both 'x' as well as 'y', are going to take the value '5', right! So, in...in this case, you need a identifier equals identifier equals expression.

So, this and this both, have to be identifiers or names or whatever you want to call them. Then, there's the unpacking assignment. What the unpacking assignment does is, it takes multiple expressions on the right-hand side and assigns them to multiple variables on the left-hand side. So, in this case, we've got two expressions or values, but they're–you can think of them expressions. And so, what's going to happen is that at the end of this 'x' is going to be '3' and 'y' is going to be '4'. That's...that's the goal there. And finally, the augmented assignment, 'x+' equals '4'. And, this is equivalent to 'x' equals 'x' plus '4', follows of course, that you can't do this without having a value already assigned to 'x'. Otherwise that's not going to work. So, let's take a look at the– this examples of this assignment here, quickly. So, we've got 'x' equals 'p' plus '10'. If you try this, we're going to get a 'NameERROR'. The 'NameERROR' says the name 'p' is not defined. And, what it's really saying is that–you know what Python does is every identifier, every symbol that you use has an entry in something called the name space, and what this is saying is, 'I can't find 'p' in the name space', essentially that's what it's saying. In the second case, of course, we have a value for 'p', So, this works nicely, right, we don't have to worry about it.

Then, in the third case, we've got the unpacking assignment. So at the end of this 'x' and 'y' are '3' and '4'. And, notice that–just note that what happens with the...with the unpacking assignment is that these are done independently. So, in the case of this, we–first, Python is going to evaluate this and assign that value to 'x', and independently evaluate this and assign the value to 'y', all right...all right! So, it's as if they are, sort of existing with whatever previous values were known to it, and it's not going to take, anything that happens inside this, into account when it does a change, right. So, that's the thing with it. So, we can...we can take this as an...as– let me show you by an example, what I mean here. So, if I take this example and say 'x' equals '5', 'x' comma 'y' equals '87' comma 'x', all right. So, what we are...what

we are saying here is that 'x' will become '87' and 'y' will become 'x'. So, let's see what happens with this. We...we should print it as well, I guess. Let's do 'print'. We find that though we would think that 'x' became '87' first, and then 'y' became 'x', so 'y' should be '87', that's not what actually happened. So independently, 'x' became '87' and 'y' became whatever 'x' was, before this particular statement was executed. The final–we have the augmented assignment and that is, 'x' equals '7' and we can plus equals to that. So, what this is going to do is, it's going to add '4' to whatever the existing value of 'x' is, and we get '11'.

Video 6 (08:30): The if statement

So now, let's briefly look at the 'if' statement because that's a very useful statement in any programming language. In Python the 'if' statement is used to control program flow, and essentially what you do is, you say, if something is true, then you do something, and if it's not true, you do something else. As simple as that. So, let's take a simple example. We have a simple trading strategy. The price of a stock drops more than '10%', then we sell the position as a 'STOP LOSS'. And if the price of the stock goes up by more than '20%', we sell the position as a 'PROFIT TAKING' kind of setup. And, if neither one or two work, then we don't do anything. So, the way this would work in Python, is that we– let's say we input these two values. So, we're using the input statement and...and introducing that as well, I guess. So, what the input statement does is, it takes an input from whatever your standard input device is. And in Python three, it always returns a string. So, if you want to use it as a number, you need to convert it into an integer.

And, the way to convert from one type to another is to use the name of the type followed by a function call. So, we are saying convert whatever value we get from this input statement into an integer. Obviously, that will work only, if it's convertible. So, you've got to be a little bit careful, All right! I mean you can't convert the letter 'A' into an integer, but you can convert the number—the letter '1', the character '1', into an integer. So, be careful about that but, that's what that...that is here. So, we get two integers 'x' and 'y'. And, we write a simple statement here that says, if the remainder of dividing 'x' by 'y' is '0', then we say that 'x' is divisible by 'y', and if it's not, then it's not divisible by 'y'. So here, there's a very simple 'if' statement. And, the format is 'if' followed by something that can take a truth value, right! That means that it could be any expression that has true or false as a possible value.

You could say, 'if x'–for example, if 'x' is not '0', that's true, right! And so, what Python will do is, if this evaluates to true, then it will do whatever comes here. On the other hand, if it doesn't evaluate to true, or if it's false, it will do what else is–what comes over here. So, the structure is very normal for programming languages. 'If'. 'Else'. All right! So, in our purchase price example, we could do the same thing. We could say get the 'purchase_price', and convert it into a floating point number. Get the 'price_now', and convert that into a floating point number. Then, check to see whether the purchase–'price_now' is fallen below '10%' or more, which means it is less than '90%' of the 'purchase_price', and that's our condition here. If that's the case, you print 'STOP LOSS: Sell the stock!', then we have a second option here that's 'elif'.

So, this is similar to case statements in other languages, except that these are evaluable to true/false and not to individual values. So, what we are saying here is, if this is true, then do this. If this is false, on

the other hand, if this were false, then check, and go here, and check to see if this is true. If this is true, then do this. And, if that is–if this is false, then do this. So else will–the else part will occur, that means hold don't do anything will occur, only if this is false and this is false, [inaudible] If both of them are false, then we get to the hold. So, it does it serially. And, the moment it finds any one is true, it does that, and then goes to whatever the next statement the– in the program is. So, let's take a little deeper example of this. So, we have here the same example, and I'll run this particular one, after we talk about it. So, in this example, we again get the 'purchase_price' and 'price_now'. And now, what we do is, we say, if our 'STOP LOSS' condition has been triggered, then we print 'Sell the stock!', and we compute our loss on the stock, right! So, we–our loss per share is the purchase price minus the current price. Otherwise, that is assuming that we haven't hit the 'STOP LOSS' price- otherwise we check and see, if we've hit our profit taking price. If we hit the profit taking price, then we do the profit taking, say we're going to sell [inaudible] profit taking and print our per share gain. Otherwise, there will be neither of these two conditions that have been hit. We want to hold the stock. And to hold–when we hold the stock, we also want to print the unrealized profit, and the unrealized profit is given by this thing here. And then, at the end of the 'if' statement, and this entire thing is the 'if', once that is over, in which- what that means is that either we've done part one or we've done part two or we've done part three, exclusively, right! We can't do more than one of those parts. After we've done those three things, we come and we run this statement, and our program proceeds as normal. So, you can think of this entire thing as just one line in the program and one complex statement. It's a compound statement, really. The key things to note here, in Python are that the syntactic element of Python is really that–what we have is a block. So, this is a block here, these two statements. And, the syntax in Python, requires that before the block, you put a colon.

The colon is important because it tells Python that what follows is a block, and it knows that it's a block, and it's very helpful because when you use a IDE, for example, when using the Jupiter notebook or if you're using some kind of, you know, PyCharm or Eclipse or anything, then the IDE will automatically handle the indenting for you. And, what is the indenting? The only way that Python knows that these two statements, this statement and this statement, belong to the same block is because they are indented the same at the end. By convention, the indent is four characters. It's a convention. It's not a requirement. But this line, say–let's say this line and this line have to be indented the same. If you add like an extra space or something after this, you're going to get a error, a interpreter error. It will tell you that it's unexpected indent error. So, these have to be the same. So, the way blocks are structured, just to note, are, there's a colon that says what follows is a block. And then, every line in the block is indented the same and indented inwards. So, we're indenting this inwards over here to differentiate this from the rest of our code, over there. That's the...the key there. And, the moment the indenting gets finished, then Python knows that that block is over, right! That's how it knows that this entire thing is one 'if' statement, because we are now going back to our indentation that matches our first two lines over here.

Okay! So this, let's say you can say on line one, statement one, statement two, statement three, and statement four. Okay! That's how that works. So indentation is, you know, really important, and it can trip you up a lot. Especially, when you are dealing with nested blocks. So, here for example, we've got a nested block, and the nested block is saying if 'price_now' is this, if 'days_held'–so this what this says is

that if we are hitting a 'STOP LOSS' price, we want to sell it, only if we've held it long enough, you know, that's the idea here. Right! So, you can think of this as a second order condition inside. So, you've got this stuff here, and clearly if you move this 'else' over here, that is, we mess up the indentation over there, then this 'else' is going to be part of this 'if', in which case we're going to have an 'elif' that is floating around, and you can get an error. Right! It's...it's going to say, "Hey! Wait a sec. There's an 'elif' without an 'if', and that's not going to work."
So, we have to make sure that we get this indented correctly over here. And, we make sure that this 'else' is indented over here. Right! So, the indentation can be kind of crucial and sometimes, for example, if this 'else' were missing completely, and we messed up the indentation over here, then we would not get a compiler error or interpreter error. It would–the syntax would be fine. However, our results would be not what we expect. So, indentation is easily one of the trickiest things in Python. If you're writing really, squirrelly pieces of code. And generally, it's a good idea to avoid doing that. If you write lots of functions, you'll be in better shape.

Video 7 (13:02): Function (Part 1)

Let's look at functions in Python. The first thing of course is, How do we call a function. The way to call a function is to use the name of the function. So, you got the name of the function 'max' over here, and then give it a list of arguments. So, in this case, we have a function that has a name 'max' and two arguments 'x' and 'y'. In Python, there is no way of specifying what the type of the arguments is, so you'll take any 'x' and any 'y' for which 'max' works and return a value. And, that value will then become the value of the expression 'max(x,y)'. And 'z', that's the variable on the left-hand side, in this case, we'll take the value of that expression. So, we have 'x' is '5', 'y' is '7', '5' and '7'. 'max(x,y)'—'7' is greater than '5', so 'max(x,y)' is '7', and 'z' becomes '7', and when we print it, we get a '7' out over there. The whole idea of the function, as we should all know, is that a function is a black box, and generally, you don't want to know what the black box is doing. You want to know what the result is. You want to know, how to call it. So, you need to know what are the arguments. You want to know what it returns, but you don't want to know what's going on inside. So, you can think of a function as a place, where something happens inside, but you don't really know what has happened and you have to, in effect, take the trust that it's doing what it says it's doing. Of course, if you've written it and tested it, you would know that. But often, you don't really know that. So when you want to use a function that somebody else had written or something that you've written, you might want to import it into your program, import a library of functions into your program. Importing libraries is very straightforward. What you do is, use...use the command 'import' and you give the name of the library, and then that name gets added to the name space, that is, a list of valid identifiers that your program understands in...in—for your program. So now, your program knows that there's something called 'math'.
What it doesn't know is what the functions in 'math' are. It doesn't know that directly, so when you want to call a function that is a part of the 'math' library, you need to disambiguate it, which means that you...you give the name of the library 'math', put a dot, and then give the name of the function that you're calling, in this case, 'sqrt', and then, you call that function. So here, square root, the function that is in the name space of 'math', and 'math' is a library that is now, because you've done the import

'math', in the name space of our program, and we can call that and run that. You can also import a function itself into the library. So, for here—for example, we could say, 'from math import sqrt'. So, what we're saying now here is that don't import the 'math' library.

So, don't do this. But from the 'math' library, import the function 'sqrt'. So, 'sqrt' gets added to our name space, our program's name space, and we can directly refer to it now. It's part of the program itself, so we—from now on, whenever we say 'sqrt', it knows that it has to call the 'sqrt' function that's sitting in the 'math' library, but it doesn't have access to any other function in the 'math' library. That's the...the difference between this method and the other method. Python is an open source library. It's an open source language, so it has many, many libraries. And many of these, in fact, most of these would need to be explicitly installed on your computer. If every one of them was already installed, you would require a heck a lot of space. So, you need to actually install them on your computer. There's a listing of all the libraries that are authenticated, so to speak. That means that they're...they're going to do roughly, what they say they will do, and they are not going to damage your computer.

At 'pypi.org' and you can go there and look at them, if you want to. But actually, installing libraries is usually a process of googling for a library and then installing it in your machine. And, the way to install that is actually pretty straightforward. You can run this program called 'pip', 'pip' is standard-it stands for the Python installer program, and what it does is, it installs a library. That means it takes the...the code for that library and installs it on your computer in a particular location, where Python can find it when you need it. So, this is going to actually download, as you can see. It downloads 'easygui' from online somewhere, and then shoves it around there and installs it and pushes it into your Python, is a particular directory, called side packages that sits in your installer, and that's where it goes. So, let's take a look at some examples of this thing here. So, let's start by just seeing a basic function 'call' example. So, here we have 'x' is '5', 'y' is '7', and 'z' equals 'max' of 'x', 'y', and we print 'z'. So, this pretty straightforward. We just call the function with 'x' and 'y', and then print 'z'.

So, what happens is that, the max of 'x' and 'y' becomes the value of 'z'. If you want to install a library, we call the 'pip' installer, so let's install this right now, 'pip install easygui'. And, in my case, it's already satisfied. And, if you notice here, it says that 'easygui' is in this directory here, which is 'anaconda', because I'm using 'anaconda Python', 'lib', 'Python 3.6', because this is the interpreter that I'm using. Every time you upgrade Python to a different version to, like, if you go on 3.6, let's say 3.7, when it comes up, then you would need to reinstall all the side packages, because this is only currently being installed in the side packages of Python 3.6. So, once we've got that we can actually use it, and the way we use it is, as we said before, you import 'easygui'. So, that's now tells us that 'easygui' is in our name space. So, we can access the function inside it, and one of the functions in 'easygui' is this function, called 'msgbox', which is-all it does is it pops up a message with an 'OK' button and you click 'OK', and you can do that. Easygui is a nice library for...for building widgets because it's really easy to set up message boxes and cancel boxes and Yes/No boxes or whatever kind-on your Windows, the whole works. It's actually very straightforward library.

So we'll run this, and I'll show you what happens here. So, I run this, and if you notice what's happened here is that it says—we've got a star over there, which means it's running. And, over here is—I don't know if you can see this, but it says-there's a black dot next to Python 3. Whenever they have a black dot over there, it means that 'anaconda', the browser is actually running something. So, you can't do

anything else, while the black dot is there. The reason for this is that somewhere behind all our current windows is the new window that's popped up, so I'm just going to do 'ctrl' 'tab' over here on my Mac. And, if you notice, there's a little feather, and that feather is Python, and if I move to that, or you can minimize your windows or whatever, then I get this message box that I put up, 'To be or not to be' 'What Hamlet elocuted.' This is exactly what we did here, 'To be or not to be', 'What Hamlet elocuted'. I click 'OK' and my program returns back to the- This is no longer black dot, so it's...it's working. And, we know there is-we can see there's no star there instead over here, instead, there's a number '4', right, so in the box that we've highlighted right now. And, the return value's 'OK', because we clicked the button 'OK', and that actually returns an 'OK'. So, that's the...the basic principle of installing libraries.

Now you can import functions. So here, let's take a look at this here. If I 'import math' and then do 'math.sqrt(34.23)', I get the square root of 'math'. I have to disambiguate it, because 'sqrt' is not in the name space of my program, but 'math' is. On the other hand, if I do 'import math as 'm' and I give it a name then, I can do this too. I didn't talk about this earlier, but 'm' is now an alias for 'math'. So now, instead of-what's happened now is that in my name space, I have the identifier 'm'. I don't have the word 'math' in the name space. It's no longer there, right, isn't it? So, instead I have an 'm', and if I want to call this 'sqrt' function, I need to disambiguate it using the identifier 'm', rather than the name 'math'. And finally, of course, I can import something directly, so here I saying 'from math import sqrt'. And, if I do 'sqrt(34.23)'- I don't need to disambiguate it anymore. 'sqrt(34.23)' gives me the...the value, the square root itself. So, these are three methods, and really, you know, it depends on what you want to do. So, for example, if all you are interested in...in the- for the 'math' library is the 'sqrt' function, then it makes sense to just import 'sqrt'.

If you have conflicting names of...of function, that means that two different libraries have the same name for a function- it can happen, you know, like initialize or something, then what you might want to do is you want to-might want to import the function-either the library, either as 'import math as m' or just 'import math', and disambiguate the function so that you don't have a conflict between which version of the function is being actually called, right! So, that's the real call functions. Of course, calling other people's functions is one thing, but at some point you might—you will probably need to write your own functions. In fact, you should get used to writing your own functions, because that's the best way to write a program, right, multiple functions. So, let's say we have a simple function here that we want to define. And, what the function should- is supposed to do is that it takes two values or two double values or two floating point values, and returns the...the percent change or the...the return if it's a stock, for example, right, the return on the asset or the investment that you've made. So, to do that we need several things, right, definitely. First of all, we need to really define the function. To define the function, we have to tell Python what to do to give you the answer to your question. So, in Python we use a keyword 'def' whenever we want to...whenever we want to define a function.

So, what 'def' does is, it says what follows is a function definition. And, just like in the 'if' statement, this is going to create a new block. Therefore, the code over here is indented. So, note the indent over here, right! It's going to be indented, because this is a block now, and all these statements are going to be part of the function. Then, we need to be able to call the function. To call the function, you need to have a name for the function, and that's this over here. So, we have the keyword 'def', which is the [inaudible] defining function. You have a name for the function, so what's going to happen is that this name is going

to get added to the name space. And, our program will now know what to do, when it sees 'compute_return' on the-in a...in a program. Then, the last thing we need is of course the arguments. So, the arguments are listed one after the other, separated by a comma. So, this is argument one, and this is argument two.
So, these are now variable names, and what's going to happen is that when the function gets called, each of these values-each of these variables, sorry, 'price_then' and 'price-now' will take the value that is passed to it by the calling statement. So, that's our basic structure of the function, of a function. We need a name of the function 'compute_return', we need the keyword 'def', and we need a list of arguments. The arguments are in parantheses and separated by commas. Python knows it's a function, because you have the keyword 'def' over there. So it knows it's defining a function. Then, you have a bunch of statements, and at the end you specify what it is you want to return. So, the idea here is that you might be doing lots of intermediate calculations.
You don't want all these calculations to show up inside the your main program. What you're interested in is the final value that gets returned, and that's what we do through the return statement. So, the return over here says that whatever variable follows this or whatever expression, rather, follows this that value is what is returned by the function. So that's how we define a function. Once we've done that, we have a function that works. And, we can now call this by doing 'compute_return', 'price_then', 'price_now' whatever. We'll see an example in a second. And, you'll get a value back for...for what the percent return on the investment or, and...and in effect actually, for any percent return, on any percent change.

Video 8 (13:57): Functions (Part 2)

The 'return' statement is what returns a value. And, if you do not have a 'return' statement inside your function, then what Python does is it returns a 'None'. A 'None', remember, is a 'None' type. A 'None' type means there's nothing, there's no value there, so it returns a 'None'. And so for example, if I have a function here, this function over here, 'def spam (x)', 'x' equals 'x' plus '1'. There's no return statement. So, when I call this function, I say, 'print (spam (5))', it's going to print a 'None' because that's what is returned by the function itself. You can return multiple values from a function. So, if I have a function, say, 'minmax', 'x' comma 'y' that returns the minimum of 'x', 'y' and the maximum of 'x', 'y'. So, what this is doing is, is returning two values, the minimum and the maximum. So, both the values come back. And then, this is–follows our unpacking assignment example that we saw earlier.
What happens here is that this function is returning two values. So, the first value 'min(x,y)' gets assigned to 'x', and the second value 'max(x,y)' gets assigned to 'y', and if you print that we get to '2' and '7' over there. So, that's how that works. And, we'll see in a second that...that might be there are some further complications that we can do with this thing here. You can pass arguments to a function. Arguments are typically passed left to right. So, if we have a function that says 'div(x,y)', over here. So we have two arguments, 'x' and 'y' and what's going to happen is, we call the function. So we call it the 'div(a,10)'. So, that's argument one and that's argument two. Then, what's going to happen is that 'a', that is, '30' is...is going to get assigned to 'x' and '10' to 'y'. So 'x' is '30' and 'y' is '10', and we get a '3'. If on the other hand, we pass– even though we call it a note, that these are the names of the variables

that are in our program, and using the same names inside our function, but they are different values, right!

They're completely different. So, what happens is that, in reality, is that when I call 'div' with 'y' and 'x' over here, so if I call 'div(y,x)' then, this 'y' over here is actually '30'. So, the '30' gets passed to here, and we get '30', not the 'y', right! The 'y' is just a name. It doesn't mean anything. So, every function actually in–if you want to know the details, every function has its own little name space, and we have mapping values from one function to the other in reality. So, that's what happens here. And, because we are passing arguments from left to right, so here, 'x' takes the value '10', 'y' takes the value of '30'. So, 'x' is '10', 'y' is '30', which means that in the function, 'x' becomes '30', and 'y' becomes '10' in the function, and we get '30' divided by '10' or '3'. You can also, in Python, give value directly to arguments and function calls.

So, for example, in my function I have here, two arguments, 'x' and 'y'. And, I can name these arguments and give them values, when I'm calling the function. So, if I say 'div', 'x' equals '30', 'y' equals '10'. So, here I say, 'x' is 30, 'y' is '10' then, what happens is that the 'x' becomes mapped to this 'x', and 'y' becomes mapped to this 'y', because I'm explicitly naming them as 'x' and 'y'. And, if I reverse the order, if I say 'y' equals '10' and 'x' equals '30' then, in this case, we are saying 'y' is '10'. So, even though this is the first argument in my function call, this is going to get mapped to this 'y', because I've named it. And, 'x' equals '30' will get mapped to this 'x', because I've named it. I have to use the names that I've given to the arguments in the function definition. So, I have to use these two names, over here to actually specify those values. But, I can do that directly. It's very helpful, actually, in many ways to do this because then you don't have to worry about the order of arguments and having to present your arguments in the right order, if you don't remember it.

You just remember that it's called 'sep' or 'end' or whatever, then you can just do that. And, in fact, most Python 'id' is that development environments will actually fill in the possible values for you. Function can have default arguments. So, in the sense, here we have 'compute_return', and we're saying that has three arguments, in this case, 'x', 'y', and 'z'. So 'x' and 'y' are just arguments, but 'z' we're saying 'z' equals '0'. And what this says is, if you don't specify the value of 'z', then assume the value of 'z' is going to be '0'. That's what this is saying. So here, for example, when we call it '1.2', '91.2', '1.2' gets mapped to 'x', '91.2' gets mapped to 'y'. And, because we don't have a 'z', 'z' is '0'. And, in this...in this case, '1.2' gets mapped to 'x', '91.2' gets mapped to 'y', and we have a 'z', so '100' gets mapped to 'z'. That's the key difference there. We also have—functions can actually have functions as arguments, and this is a phenomenon, or some—something that is called, first order functions in programming languages. And, what that means is that functions and variables are considered equivalent.

They are the same—their... their functions are also objects, and in Python, there are also objects because in Python everything is in the name space, equally, right! So, we have—'x' is something, 'y' is something, and 'order_by' is also something. They're just lists of things in our name space. So, there's no difference between one and the other. So, that's why they're called, first order functions, and Python allows for first order functions. So, here for example, we have a function called, 'order_by', and what 'order_by' does, is it takes three arguments, 'a', 'b', and 'order'. And, notice that we haven't specified what they are, but Python will figure that out, because 'a' and 'b' are numbers and 'order' is a function. And, how...how will it know 'order' is a function?

Because in...in–when it calls it, the name 'order', that's the identifier 'order', is going to be followed by an open parentheses. So, whenever you have any identifier followed by an open parentheses, then Python assumes it's a function. So, that's going to be a function. So what's going to happen is, that when I call 'order_by' with '4', '7' and 'max', then 'max' is going to be mapped to 'order', the name 'max', '4' to 'a', and '7' to 'b'. And, when I do 'order_by'– when I do 'order(a,b)', this becomes equivalent to 'max', where order is replaced by 'max' and '4', '7', because that's what this over here is, 'order(a,b)', all right! So, that's how the function works as a first order function. So, let's go ahead and take a look at some examples and see how that works. And you should, you know, always try all of these things. The best thing to do is when you're trying these things is to play around with them, change a few values, try putting your own stuff over there, and see what happens.

That is always the best way to find out what's–what's goes on inside a piece of code is to, you know, mess with it. When it breaks, you know you've done something wrong. When you fix it, you learn something. That's how one learns programming. So, here's a simple function 'spam(x,y,k)'. So, what this does is it returns 'p'–and the value of 'p', which is 'z' divided by 'k'. And what this– the function is unnecessarily complicated I would say but, it says if 'x' is greater than 'y' then, it returns 'z' equals 'x'. Otherwise, it returns to 'z' equals 'y' and then divides 'z' by 'k', okay! So, we've got—it's either defining 'z'—dividing 'x' or 'y' by 'k', depending upon whichever number is greater. So, that's a function, we call it with '6', '4', '2', and we get a '3.0', all right! If there is no return statement, we get a 'None', right! So here, we do 'print(eggs,4,2)' but, there's no return.

We return—there's no return value here, so it's just returning a 'None'. You can return multiple values. So, we have a function here called, 'foo(x,y,z)'. And, 'z' tells you whether you want to-it has two options, descending or ascending. If it's 'z' is descending, we return the 'max' and then the 'min' of 'x', 'y', and then we return the value of 'z', whatever it is. If the 'z' is ascending, we return the 'min' first, then the 'max', and then the value of 'z', whatever it is. Otherwise, if it's neither ascending nor descending, we just return them as is. So here, what we do is, we're returning three values 'x', 'y', and 'z'. So what we're going to do– however, on the left-hand side of the assignment statement, if you have an assignment statement that this returns to, we would have three values 'a', 'b', and 'c'. So this is what we have here. So, what's going to happen is that this one returns three values, 'a' will take the value of 'x', 'b' will take the value of 'y', and 'c' will take the value of 'z'. And we get '2', '4', 'ascending'. So, if there's only one value, then it will unpack it.

So, what Python is doing? Remember, this is an unpacking assignment, it's actually unpacking the three values returned by this, and assigning them to each of 'a', 'b', and 'c' in turn, right- each of 'a', 'b', and 'c' individually. If you don't have return–if you don't have three values on the left-hand side, then what's going to happen is that you will get the values un–not unpacked. They will be packed values. And, that's what we're getting over here. This is a data type called, tuple, which we'll see later, but is really a collection, right? So, it's not–it's the value returned is...is this collection of '2', '4', and 'ascending'. It's not each of them, individually. You have to make sure that your values are properly matched. So, here for example, I have two values on the left-hand side, 'a' and 'b', but my function returns three values, and that's a problem.

It can take three values and put it inside one value, the way we've done over here. It can take three values and put them inside three values, the way we've done over here. But, it can't take three values

and put them inside two values, that–what this is requiring. So, what we get instead, is a value where, it says too many values to unpack, okay! So, that's the–watch out for that. You should know the right number of values for this thing here. Values, arguments are in left to right, and we can see that here. It's 'x', 'y'; 'bar', '4', '2'. So, 4 gets–'x' becomes '4', 'y' becomes '2'. If you want to assign them, specifically, we could say 'y' is '4', 'x' is '2'. So this way, we've done a half because in our case, 'y' takes the value of '4' and 'x' takes the value of '2', because we have explicitly assigned them. And, we can see the function arguments as well, like we saw before. So, we have two calls to this function, the 'order_by' function that we saw before, and we'll be– what we're doing is we're saying, all right, we are going to first find the minimum of '4', '2' and then the maximum of '4', '2'. And we do that, in the first case, we get '2', which is the minimum of '4' and '2'. In the second place, we get '4', which is the maximum of '4' and '2', okay! So, that's the deal there.

Notice that if we–the way Python works is that the–though when we're calling this function, we've named the function 'order_by', we are passing it in 'order' function, so we're assuming that this is going to take a 'max' or a 'min', however that's not really required, because Python doesn't really care what you're passing to it. What it does instead is, it tries and sees will it work or not. So if I did something like this, 'order_by("hello","goodbye",print)' That works! It works because all that's really happening is that the value 'a' is getting the value 'hello', the value 'b' is getting the value 'goodbye', and the value 'order' function is getting value 'print'. And then, it goes into the function, and that function just calls 'print(a,b)', which works fine, and it returns that, right!

So, that's the deal there. So, some things will work, some things won't. Let's take another example. So let's say, I did 'order_by("hello",54, max)'. That doesn't work. Why doesn't it work? It doesn't work because I can't— I'm calling the function 'max' here, and I'm saying, give me the 'max' of 'hello' and '54'. But, 'hello' and '54' are not comparable. There's no method of comparing the two of them. So, that's not going to work. However, if I just replace the 'max' by a 'print', so let's do that here. That works! The reason is that I'm not really comparing 'hello' and '54', anymore. I can print, in the same print statement, I can pass a string as well as a...an integer, and it will print them, and so that worked. So, what Python really does is it tries it. You know, you won't know if something doesn't work until it doesn't work. So, you want to be a little bit careful with that.