Module 2: Programming with Basic Logical Structures

Video Transcripts

Video 1 (00:53): Unit Overview

Hi everyone. Over the next few modules, we'll discuss how to build the control flow of a program using different kinds of statements. This will let us build programs that take different actions based on the data provided, instead of always running the same set of statements. In Module 2, we'll expand the concept of a function from module 1 and learn how functions interact with the program's execution.

We'll also introduce a new type of data, a Boolean and a new type of statement, a conditional, which will let the program make a choice between multiple actions based on provided data. Finally, we'll discuss how to use debugging strategies, when your code causes an error message or doesn't work correctly.

Video 2 (03:34): Advanced Functions - Returning

Previously we learned how to write a basic function definition that represented an algorithm and code. We were able to call a defined function on different inputs and the functions parameters were assigned to those input values. The function call executed the lines in the functions body until the return statement was reached. The call then evaluated to the value returned by that statement which is the output of the function. Now, a return statement marks the end of the algorithm in the function body.

When a function call reaches a return statement, we say that it exits the function. That means that if you add a line of code directly after a return statement, the call will never reach it. For example, in this code, "Maybe" is never printed. For now, the return statement should always be the last line of the functions body. Now, what happens if we don't put a return statement in a function body at all? When we call the function, Python will still execute each of the statements in the functions body in order.

If Python reaches the end of the function body where the indentation stops and it doesn't find a return statement, it will send back a special data value called "None" to the original function call. In the example here, because the function double doesn't have a return statement, it will return none automatically. When you type "None" with a capital 'N' lower case 'o-n-e' and no quotes, it has a special meaning in the Python syntax, it effectively means there's no value here. It even has its own special type, the NoneType.

Now, why do we need them? Well, every function call must eventually evaluate down to a single data value, sort of like how two plus two evaluates to four. For example, in the code shown here, double (5) evaluates to 10. If a function call never received value from the algorithm, it would be

like having a hole in the code where the functions calls value should be. We use none to plug that hole so that the code can always continue running.

We can also return none by writing a return statement with no value after the return or by explicitly writing return none as shown here. Now, we're talking about return statements, let's address a common misconception. Many new programmers get confused about the difference between print and return and aren't sure when to use the two constructs. Now, print statements can be used anywhere in code but return statements can only be used inside function definitions, but what else makes them different?

In general, you should only use print to display values to the interpreter. Displaying values is definitely useful but those displayed values can't be used in future computations. For example, if we print greeting instead of returning it in our function say _hi, we can't store the functions result in a variable and combine it with another string later on. To store the value produced by a function, you need to use a return statement to send it back to the original function call.

Whether the value is then stored or modified or even printed depends on what the code does with the function call itself. When we return greeting, it becomes possible to concatenate another strain on to it and make a new sentence. You should almost always make sure to return a value from your functions. Don't just use print.

Video 3 (02:45): Advanced Functions—Parameters

We can design more advanced functions by updating the function parameters as well. So far, we've assumed that the function call will always provide an exact number of parameters. But there might be times where we want to write a function, where one or more of the parameters are optional. We can use keyword arguments to give the parameters in a function definition default values. And these look-like normal parameters, but there followed by an = sign, then the default value that is provided.

In this example, the keyword argument "title" is given the default value of an empty string, two quotes with nothing in between. In this example, if the function is called with two arguments, they'll both be sent to the provided values as usual. So, if I call, say_hi on Carnegie and Mr., it will return Hi, Mr. Carnegie. On the other hand, if the function is only called on one argument, the keyword argument is assigned to the default value instead.

So, if I call say_hi on just Carnegie, It's the same as calling, say_hi on Carnegie and the empty string and both would return hi, Carnegie. You can also directly refer to the parameter by its name in the function call, when the parameter is a keyword argument. So, if we call say_ hi on "Carnegie", title= Mr. It's a bit like calling, say_hi "Carnegie" Mr. directly. This is mainly useful when you have a lot of keyword arguments and you want to mix up the order they're provided in.

But be careful with regular parameters, because they always have to be provided in their original order. Another interesting property of functions is that they can be designed to take any number of arguments, instead of a set number of arguments. For example, print can take multiple

arguments and print them all on the same line separated by spaces. Print also has some keyword arguments. "Sep" specifies the character that should be used to separate the printed arguments and defaults to a space.

In this example, if we change sep=",", will get one, two, three, instead of one two three. "End" specifies the character that should be put at the end of the printed text and defaults to the character that is added when you press the enter key, which will call a new line. In this example, when we change end to be -, to print commands can display on the same line. You can use as many or as few keyword arguments as you want in a function call, any keyword arguments that aren't used, just default to their usual values.

Video 4 (02:11): Library Functions

Python has a large number of functions that are used for special purposes. We've seen some of the built-in functions already, like print() and round(). Most of these functions are organized into libraries to minimize the number of functions that Python has to keep track of. If you want to use a function from a library, you have to import that library first to give your program access to it. To do this, use a special statement called import followed by the name of the library that you want to load.

We'll learn about many libraries during this program. For now, let's just focus on one, the random library. To call a function from a library, we have to preface the call with the library's name to show where the function comes from. For example, the random library contains the function randint, which chooses a random integer in the range [a, b], inclusive. a and b in that case are the functions arguments. To call this function after importing random, you must use "random.randint".

Another functional library is random, which generates a random float in the range from to 1. Again, we need to call this with the library's name first followed by a period. It's a bit confusing that both the library and the function have the same name, but Python will still figure it out. If you want to skip the library name, you can use a slightly different syntax to import the library. From library_name import function_name, loads the chosen function directly into the program without needing to reference the library's name again.

For example, from random import randint, lets you call randint directly later on. Now that you have the random library, you can start writing some interesting code. Random libraries make it possible to implement all sorts of games and fun little programs. Try thinking about what kinds of interesting programs you could write by using random.randint and random.random.

Video 5 (05:24): Interacting Functions

We've now covered all the syntax of function definitions and function calls that you'll need, but we haven't discussed how functions really work behind the scenes. Function calls follow order

of operations like other built-in operations. For example, if we define a function that doubles the given argument, then run (5 + double (3)), Python has to evaluate double (3) before it can evaluate 5 + 6. However, function calls are often more complicated the built-in operations.

A function might execute several statements before it reaches the return statement and that means that Python needs to keep track of where it was before in the code while it runs through the body of a function. So, consider the example code that's shown here. When Python executes this code, it will first see the def foo (x) line and it will load that function into memory. It doesn't run the function yet though because it's still abstract, it hasn't been called on a specific argument. When Python reaches the y = 10 line, it stores 10 in the variable y.

Next, Python goes on to the y = 20 - foo (y-2) line. To execute this line, Python first needs to evaluate foo (y -2), which is foo of 10 -2, which is just foo (8). To call the function, Python will need to store the y = 20 - foo (y -2) line in something called a stack. Then go back to the function definition and assign x to 8 to provide an argument. This is an example of control flow.

As soon as Python moves back to the function definition, it's no longer going through every line of the program sequentially, it's changing the flow of the program to be something new. There will be a lot more examples of code that changes the control flow, which we'll learn about soon. So, going back to the function, Python runs through the function definition, executing each line in order until it reaches the return.

At this point, x = 19.0, so 19.0 is returned. To figure out where the value 19.0 should be sent back to, Python looks at the stack again and it finds that y = 20 - foo (y -2) line is there and goes back to that line to continue the previous work. In other words, the stack is a bit like a bookmark, it shows Python where to go once it's finished with its function call. foo (y -2) is replaced with 19.0, so the line becomes y = 20 - 19.0. This evaluates to y = 1.0, so 1.0 is stored in y. Now, there are no more statements to run, so the program is finished.

It isn't that hard to trace how code executes when there's a single function, but things get more complicated if you have a function that calls a different function in its own body. So, consider this next example. The function outer calls the function inner inside its own body. That will add another level to the stack of prior locations that Python needs to keep track of. But that's okay. Python can keep track of as many function calls as it needs to. Let's trace to this code together. First, Python will load outer and inner.

Then it reaches the line print (outer (4)). Python needs to call outer on four, so it puts this line on the stack. Now, Python starts running through the outer function with x initially set to 4, y is set to 2.0, and then we reach the line return inner (y) + 3. We now need to call inner before we can execute the rest. So, Python will add this line to the stack 2. Now, Python starts running through the function inner with a initially set to 2.0, which was the value held in y, b is set to 3.0 and then b is returned.

When Python needs to decide where to return, it will look for the most recent location that it bookmarked on the stack. In this case, that's in the outer function. So, Python moves back to

that location and then it removes it from the stack. We'll replace inner of y with 3.0, which was the return value of that call to get return 3.0 + 3 for a return value of 6.0. Again, Python looks at the stack and now it sends the value back to print (outer (4)). Now, Python can finally print 6.0 and be finished with the code.

You can actually see the stack of locations being tracked by Python when errors occur in nested function calls. Let's introduce an error into our code to show what this looks like. The print (oops) line will cause an error because (oops) is not a defined variable. So, what does this error message look like? Well, look at the indented middle of the error message. This part of the message shows the current state of the stack when the error occurred. When reading and comprehending code, you should try to keep track of function calls in a similar way.

Video 6 (05:03): Variable Scope

One last thing to keep in mind when working with functions is how they interact with variables. When you define a function, you're describing, how that function will work on a hypothetical input. That means that any variables defined inside the function or hypothetical two. They won't be assigned values until the function is called. Because of these variables, inside a function definition cannot be accessed outside of that definition, we say this is due to a variables scope. There are two main variable scopes; local and global.

A variable has local scope, if it's defined inside a function definition. In the function say_hi, shown here, both name and message are local. If you try to access a local variable outside of the function it belongs to, you'll get an error. For example, if we try to print message after calling the function, we'll get a name error. A variable has global scope, if it's defined outside of a function with no indentation, we call this the top level of the code. In the example shown here, user is defined outside of say_hi, so we can safely print it at the top level.

We can reference global variables at the top level, but we can also reference them inside of functions. This happens when a function body references a variable that has not been defined in the local scope, Python checks if there are any variables in the global scope that match the name. In the example shown here, Python does not find a local variable greeting and say hi, so it checks at the global level and uses the greeting variable that it finds there. This is like a two-way mirror.

The top level of the code cannot look inside a function definition to see its variables, but a function can look outside of itself to see the global variables. Now, what happens if the same variable name is used both locally and globally? Like in the code shown here? Well, these two variables might have the same name, but they're separate, and they can have different values. It's like knowing two people both named Andrew.

They have the same first name but different last names in the code here, the last name of the first greeting would be global, and the last name of the second greeting would be say_hi when there are multiple names to choose from. The function will always default to its local variable.

So, in this case Python chooses the say_hi greeting and prints "Hi Carnegie", instead of hello Carnegie. Now, remember parameters count as local variables too.

For example, the code shown here, even though there's a global name variable, the local name parameter will still be used first. You might occasionally want to change a global variable inside a function, to do this, you have to use a special statement to keep the function from creating a new variable with local scope inside of the function. For example, in this code because greeting is assigned directly to evaluate the function, there will be two greeting variables.

One local, one global. Changing the local greeting variable does not change the global greeting variable at all. To fix this, use the global statement, before assigning the variable to a new value. This tells Python to look for a global version of the variable instead of making a new local one. And now the function can change the global variable directly. Finally, note that variables with local scope are local to their particular functions. If your code has more than one function, each function will have its own particular scope.

So, let's consider this code. There are two variables with the name greeting. One global, one local to the leave function. When greeting is referenced in the leave function, the local version is used but when greeting is referenced in the arrive function, it can't access the greeting variable and leave. So, the global version is used instead. There are also three variables with the name, "name", one global, one local to arrive, and one local to leave. Each function will refer to their own local variable while inside the function body.

The top-level code will refer to the global name as it cannot access either local variable. Note that one name is changed in the arrive function, it does not change the value held in name at either the global level or the local level of leave. The three variables are separate. Scope might be confusing at first, but it is important to keep track of which variables you have access to in different parts of your code. For now, you can make things simpler by avoiding global variables most of the time, and by using well named variables to keep your code readable.

Video 7 (08:06): Boolean Values and Operations

Next, we'll talk about a new data type that will help us make decisions based on data and code. This will open up exciting new possibilities for the kinds of programs we can write. This data type is called a Boolean, or bool for short. Booleans are different from integers, floats, and strings because they can only hold one of two values, 'True' or 'False'. In other words, Booleans measure correctness or truthiness. Why do we call these values Booleans instead of truth values?

It turns out they're named after George Boole; a mathematician who introduced many of these concepts back in the 1800s before computers were made. When you type "True" with a capital T or "False" with a capital F, these terms are evaluated as Boolean by Python instead of being read as variables. More often, we evaluate an operation or on other data values that reduces down to 'True' or 'False'. Python can perform a variety of comparison operations that compared two values.

These comparisons are similar to what you've already seen in math courses. The operation evaluates to 'True' if the comparison is correct, and 'False' otherwise. For example, '2 < 4' evaluates to 'True' because two is less than four. '4.5 >= 10' evaluates to 'False' because 4.5 is not greater than or equal to 10. We can also evaluate 'greater than' and 'less than or equal to' operations as needed. Python can also compare two values to see if they are equal or not.

I can't use a single equal sign for this, though, as those symbols already used for variable assignment; instead, we use two equal signs, or double equals, to compare for equality. To check whether two values are not equal, we use an exclamation point followed by an equal sign, and we usually just call that "not equal" directly. So, '(2 + 4) == 6' evaluates to 'True'. And '9!= (10 -1)' evaluates to 'False'.

Be careful when you're comparing floating-point numbers for equality. Floating-point numbers are not represented in Python with perfect accuracy, because some numbers, like 1/3 for example, can't be represented in a finite number of digits. This leads to some odd errors. For example, '0.3 + 0.3 + 0.3" should equal 0.9. But in Python, it doesn't. It equals 0.899999 etc. instead. That means, that if you check whether '0.3 + 0.3 + 0.3 == 0.9', it will evaluate to 'False'.

To account for these rounding errors, it's safer to check whether two floating-point numbers are almost equal. That is, equal within a certain error bound. Python has a function that does this for us already. Import the built-in math library, and call 'math.isclose' on the two values being compared. This puts the error bound a 10 to the negative 9th by default. The math library has plenty of useful math functions and values, but we'll just use 'math.isclose' for now.

We've only compared numbers so far, but we can also run comparison operations on strings and other types. For ordering, we won't worry about it too much for now, but we will want to use equality checks on strings at some points. Note that just like with mathematical operations, Python doesn't like performing comparison operations across different types. You can check whether two different types are equal, but if you use 'less than', 'greater than', 'less than equal to', or 'greater than or equal to' on two different types, you will get an error.

Just like with mathematical operations, we can use order of operations to combine Boolean operations and values into more complex expressions. But first, we need a way to combine two or more Boolean values together. We'll now learn about three core logical operations that can combine Booleans together. The first is the 'and' operator. 'And' takes two Boolean values or expressions and evaluates to 'True' if both values are 'True', and 'False' otherwise. In other words, if either the values are 'False', then the whole thing will evaluate to 'False'.

'And' is useful when you want to require that two conditions both be met at the same time. For example, the code shown here can check whether a number has exactly one digit or exactly two digits. The second is the 'or' operator. 'Or' takes two Boolean values or expressions and evaluates to 'True' if either of the values is 'True', and 'False' otherwise. In other words, 'or' only evaluates to 'False' if both of the values are 'False'. 'Or' is useful when there are multiple valid conditions to choose from.

For example, the code shown here can check whether a given day is during the weekend or is a specific weekday. Our third operator is the 'not' operator. 'Not' takes just a single Boolean value and reverses it; 'True' becomes 'False' and 'False' becomes 'True'. Not as useful for switching Boolean results when the opposite of what you want to check is somewhat easier to express and code. For example, the code here can check whether the given number is non-zero.

Now that we have these logical operators, we can combine them with comparison operators to make more advanced expressions. Take a look at this example expression. Pause the video for a moment to see if you can figure out what it will evaluate to. Well, the values and parentheses go first, so we should evaluate '(x > 5'), or '10 > 5', which is 'True'. We can also evaluate '(10**2 > 50', which is 100 is greater than 50, and that's also 'True'. So, now we have 'True' or 'True', and '(x == 20)'. True or true just evaluates to 'True'.

And '(x == 20)' will become '(10 == 20)', and that evaluates to 'False'. So, our final expression is 'True' and 'False', and that finally evaluates to 'False' overall. There's one final thing you should know about logical operations. When Python evaluates a logical expression, it's kind of lazy about it. Python will only evaluate the second half of the operation if it needs to. And this is called short-circuit evaluation. For example, if you're checking 'x' and 'y', consider what will happen if 'x' is 'False'.

Well, one possible outcome is 'False' and 'True', and that will evaluate to 'False'. The other outcome is 'False' and 'False', and that will also evaluate to 'False'. Either way, the whole expression will be 'False'. That means that the second value kind of doesn't matter, and Python actually won't even check it. You can put code into that second value, which should cause a runtime error. But because it doesn't run, no error is raised. Likewise, if we're checking 'x' or 'y', if 'x' is 'True', the expression will always evaluate to 'True'.

Again, in this case, the second half just doesn't run. Short-circuit evaluation will be useful in cases where we need to run code that might cause an error if a certain condition isn't met. Just use an 'and' operation with the condition in the first half. If the condition is 'False, the second half will never run. For example, in the code shown here, we want to compare two values. But this will cause an error if they aren't the same type. Use short-circuit evaluation to catch the error before it happens.

Video 8 (06:01): If Statements

Now that we have Booleans, we can use those 'true' and 'false' values to make decisions in code based on data values. Instead of always running the same set of statements, Python will be able to choose which of a set of actions to execute based on the current code state. This opens up the possibilities for the kinds of algorithms we can write a great deal. To do this, we'll learn about a new control structure called a conditional. The most basic form of a conditional is an 'if' statement. An 'if' statement has a structure shown here.

Like a function, the structure puts a colon at the end of the top line, and contains a body, which is indented. The expression immediately to the right of the 'if' keyword must evaluate to a Boolean. If that Boolean is 'true', Python runs through the code inside the 'if' body normally. If the Boolean is 'false', the conditional body is skipped entirely. For example, the code shown here checks whether the variable 'x' is less than 10. When 'x' meets that condition, as shown here, the code will print "hello," "wahoo!" and "goodbye."

On the other hand, if 'x' does not meet the condition, as it shown now, only "hello" and "goodbye" are printed. Just like with function definitions, the conditional body can have as many lines as it needs. The end of the body is marked by the end of the indentation. So, in this code, if the word is "go," the lines "1... 2... 3!" will all be printed before it prints "finished." If the word is something else, we'll only see the word "finished" be printed.

If you want a program to choose between one of two possible actions, you could write two opposing 'if' statements, as it's shown here. But there's actually a better way. You can use an 'else' statement connected to an 'if' to provide a direct alternative. The 'if-'else'' and the two bodies are all treated as a single control structure instead of two different conditionals. This is useful when you want to think of two possibilities as being alternatives. If the program does not enter the 'if' body, it will immediately enter the 'else' body instead.

And if the program does enter the 'if' body, it will skip the 'else' body entirely. Only one of the two sets of statements gets to run on any given program execution. Finally, if you want to have a program to choose between three or more choices, you can add an 'elif' branch to the conditional structure. Again, this is part of the larger control structure. This works just like a normal 'if' statement, except that you only enter the 'elif' branch if the 'if' branch that came before it evaluated to false, and if the 'elif' Boolean expression evaluates to true.

For example, the code shown here only prints "small" if 'x' is equal to seven, because the first expression would be 'true' and it wouldn't even look at the rest. If 'x' was equal to 23, the first expression would be 'false' and the second would be 'true', so "medium" would be printed, and then it would skip the 'else'. To print "big," 'x' would need to fail both of the tests. So, it would have to be greater than or equal to 10, and greater than or equal to 100.

In a single conditional control structure, you must have one 'if' branch and you can only have zero or one 'else's, but you can have as many 'elifs' as you want. For example, this program prints out different responses based on the day of the week. Using an 'if', three 'elifs', and an 'else'. The program will only ever print out one of the responses, because all five branches of the conditional are connected into one structure. Now, so far, we've only looked at conditionals that were written at the top level of the program.

But they can be written in other places, too. For example, we can use a conditional inside a function definition if we want to. To do this, just start the 'if' statement at the same level of indentation as the rest of the function body. Then, indent the body of the 'if' even further so that it's clearly at a different level. An example is shown here. We call this nesting, a conditional inside

of a function. Note that when you nested conditional in a function, you might want to use more than one 'return' statement.

And this is actually okay, because Python will still only reach one 'return' statement overall in a single function call. We're just changing which statement is reached based on the data values. Python will still exit the function immediately after executing a 'return'. Because of this, we can sometimes get away with just using an 'if' instead of an 'if' paired with an 'else' if we 'return' inside the body. So, consider the new approach to find average shown here. If the function goes in the 'if' statement, it will return early and will never reach the last line.

If the function doesn't go in the 'if' statement, it will go immediately to the last line and execute it normally; just as it would have done if that line wasn't an 'else'. So, this approach implements the exact same logic, just with slightly less code. You can use nesting to arrange more advanced control flow in a program, as well. So, for example, you can nest a function inside of a conditional; though, you'd really want to. And you can nest a conditional inside another conditional. This makes it easier to check complex conditions.

Note that when you nest conditionals, the 'elifs' and the 'else' will always be paired with the 'if' that's at the same indentation level. For example, in the code shown here, the 'else' will go with the 'if' labeled with an "A" since they're both at the outermost indentation level. If we want to add an 'else' to the 'if' labeled with a "B," we have to put it before the outer 'else'. Because if we don't, the inner conditional will be interrupted by that outer branch. We'll go over more examples of when you might want to use nesting in the next video.

Video 9 (07:24): Problem Solving with Conditionals

Let's look at two examples of how we can use conditionals and Booleans for our problem-solving. First, let's write a simple bit of code to check whether a library book is late. And if it is late, to calculate how much the late fee is. First, I'm going to write a little bit of code to check whether a library book is late. If it is late, I want to print out a message about that, as well as the late fee. If it's not, I can just print out a short message instead. First, I should set up the variables that we'll need.

We need to know the maximum number of days that you can have a book checked out. So, I'll go ahead and set up a 'max_days' variable, and let's just set that to 30. And I'll also need to know what the late fee is per day. So, I'll set up a variable 'daily_fee', and let's just set that to 10 cents to make things simple. I'm also going to need to know for each specific run of the program, how many days that specific book has been checked out. So, I'll go ahead and set up a variable 'days_passed', and for the first run, I'll just set it to 55.

We can change it later to see what happens. Now, I need to look at two possible situations. One is that the book I've checked out is late and has a late fee. The other is that the book is not late; I'm still fine. Now, in that second case, this will happen if somebody has the book checked out for 30 or fewer days. So, I can say 'if days_passed <= 30'... Well, in this specific situation, now

we're fine. So, I'll just print it out a little statement that says, "You're fine!" Very nice. Now, otherwise, I'm in a situation where I'm past that 30-day mark, and I am late.

Because I know this will handle all the other values for 'days_passed', I can just use an 'else' statement right here to say otherwise. I'll figure out whatever the total fee is. And then, I can print a little statement that says "Late! Total fine:' is some variable 'total_fine' that we'll set up. To calculate how many days have passed since the end of my book's check out period, to do that, I can take my 'days_passed' variable, and then subtract the 'max_days' period.

The number of days that have passed since the maximum of days is the number of days that it's late. So, I'll take that total number of days and I will multiply it by the daily fee, and that should get me my total fine overall. All right. That looks good to me. Let's try running this and to see if it works. Now, if I had the book checked out for 55 days, well then subtracting 30 from that, I should have five days late. And 25x10 should be 250.

Let's try to run it. And indeed, it says that it is late, and the total fine is 250 cents overall. If instead I checked out the book for 31 days, that I would only be one day late. So, if I run it again with this. Now, I only have a 10-cent fine because I'm only one day late. I should try this with some values that aren't late, as well. So, for example, I can try this with the book that I've only checked out for 17 days. And then, it will say, we're fine in that case. Works. Now, for the second example, now I want to do something a little bit more complex.

I want to write a program. The program that I will call 'can_rent_car'. Let's do this one in a function just for fun. And this program will take in some attributes of a person. In this case, their age as an integer, and whether or not they have a license as a Boolean. And it's going to determine whether or not we want to rent them a car. And what we're going to say is to be able to rent a car, you'd have to have a license, and you have to be at least 26 years old for insurance purposes. Now, there's two ways I can go about solving this problem.

I can either do it with nesting or I can do it with a logical operation. I'm going to try the nesting approach first, and then we'll look at how to change it into a logical operation afterwards. For the nesting approach, I got two things I want to check. I want to check whether 'age' is at least 26, and I want to check whether 'has_license' is 'True'. For this, I'll go ahead and start with the age check. I'll say if age is at least 26, is greater than or equal to 26.

Well, if it is, I want to go and do some checks on the license. But if it isn't, it doesn't really matter whether the person has a license or not. They're not going to get to rent a car. So, right away, in this 'else' case, I can just go ahead and return 'False' as my output. In the 'if' case, I want to do another conditional. And note that I'll indent this inside of the first one to show that it will only happen if we get in that first condition.

So, we'll say 'if has_license' is 'True, well then in that case, we meet both of the requirements. We can say, yes, we'll rent you a car, we'll return 'True'; otherwise, if you are old enough, but you don't have a license, we'll go ahead and return 'False. So, we can run this. And now let's try testing

it out. So, first, we want to rent a car to somebody who is old enough and does have a license. So, maybe somebody who is 27 and does have a license. Yes, we can rent them a car.

Well, what if someone is 27, but does not have a license? No, we're not going to rent to them. Similar if we look at someone who's 23 and does have a license; we don't rent to them. And somebody who is 23 and does not have a license, definitely not going to rent to them. So, this seems to be working in general. Now, I mentioned this is the nesting approach, but it looks a little repetitive with all those 'return' statements. So, if we want to, we can cut down on this repeated code by using logical operations instead of nesting.

For example, in here, I'm saying 'if has_license return True else return False'. Well, this is all really just the same thing as saying 'return has_license'. Think about why for a second. If I'm returning 'has_license' in here, if 'has_license' is 'True', that's the equivalent of going into the 'if' statement right here where I would return 'True'; otherwise, if 'has_license' was 'False', then I would have gone to the 'else' statement and return 'False'. So, this is doing exactly the same operation.

And indeed, if I run this and try some of my test cases again, we can see that it still works properly. And now, I can do another combination of the two parts if I want to get it all down to a single line instead. So, what I was saying here is, well, if I'm going to return 'True' at all, I'm only going to return 'True' if 'age >= 26', and if 'has_license' is 'True'; otherwise, I should return 'False'. So, let's go ahead and take this 'age >= 26' and move it into this 'return' to say, 'return' if 'age' is greater than or equal to 26, and 'has_license' is 'True'.

So, this whole thing right here does the exact same operation as our original 'if' statement. If I run this and try my two test cases again, they still work properly. Now, you won't always be able to condense logic this way, but it's useful to do so when possible, to keep your code just a little cleaner and easier to read.

Video 10 (03:26): Debugging Syntax/Runtime Errors

In this next section, we'll discuss a core part of the programming process. Fixing your code when it doesn't work. Programming errors or bugs as they're commonly called are extremely common even for experts. It is perfectly normal to occasionally make mistakes when you're writing code, just like how it's normal to have typos when you're writing text. But when you're writing code, even a small error can result in the code not working at all. Learning how to find and fix errors is therefore extremely important.

Previously, we've discussed three different types of errors, syntax, runtime and logical errors. Both syntax and runtime errors show up in the interpreter with detailed error messages. This gives you a lot of useful information about what's going wrong in the code. Here's an example of a syntax error. Now, that the bottom line of the text says syntax error and that's usually what syntax errors will be called. Though you may also see indentation error or a message about the code being incomplete.

Syntax errors give you the line number where the error occurred in the file, which is often though not always accurate. They also include a little in line error that points to where in the line python thinks the error occurred. Again, this is often but not always accurate. Here's an example of a runtime error. Runtime errors also have line numbers and those are usually quite accurate. Runtime errors also have distinct names and descriptions on the last line of the error message. We've seen examples like name error and type error.

When your code causes syntax or a runtime error. The best thing you can do is read the error message carefully. Start by looking at the line number to see where approximately the error occurred. Next, look at the error type to see if you're dealing with the syntax error or a runtime error. If it's a syntax error, look closely at the code indicated by the in line error, it's likely that something is a bit off about your syntax there. If you can't find the problem, try comparing your code to example code that you know works.

If it's a different type of error, read the last line carefully to see what the type of error is. This will help you narrow down which part of the code you need to look into a great deal. For example, if the message says type error then you know, there's a mismatch type somewhere. You can consider the types of your data values without having to worry about anything else. Let's look at an example together. Take a moment to pause the video and review the error message generated by this piece of code.

What do you think the problem is? Well, the error occurs online three. So, we know the code ran successfully until it reached the print line. The error type is a name error that means the code reached a variable that it could not identify. The errors message specifically mentions the variable name, friend. Looking at the code more closely, you can see that the original variable was spelled IE but the variable in the print line is spelled EI. It's a typo. Fixing the typo will fix the code as well.

It can often be harder to notice typos and small mistakes when you're reviewing code that you've written yourself. If you get stuck with debugging syntax, or a runtime error, consider taking a break and coming back to the code later. This will often help you see the code in a new light.

Video 11 (04:37): Debugging Logical Errors

Now, let's discuss how to go about debugging logical errors. As a reminder, logical errors occur when your code is able to fully run but doesn't do what it's supposed to. In this program, you'll mainly encounter logical errors when your code fails a test case on an assignment. Test cases are implemented using assert statements. These are special kinds of statements that take a Boolean expression as a value. If the Boolean expression evaluates to True, nothing happens.

If the expression evaluates to False, the code raises an Assertion error. When your code raises an assertion error, the best thing you can do is examine the input, the expected output and the actual output of the failing test case. The expected output is the value being compared to the function call in the assert statement showed in the error message. To get the actual output, you

should copy the function call and just the function call from the error message and run it in the interpreter.

This will show you what your code is actually doing. If the expected output seems incorrect to you, you've probably misunderstood the problem statement. Go back and reread the prompt to see what you've gotten wrong. If the actual output seems incorrect, compare the expected output to it to see what the main difference is. You can then use a debugging process to investigate your code and see what's going wrong. There are several debugging processes that can help programmers identify errors.

We'll go over a few possible approaches now. The first approach is called rubber duck debugging. You want to find a person, pet or inanimate object like a rubber duck and explain to them how your code is supposed to work. In the process of explaining your code out loud to someone else, you may find that a piece of code doesn't match your intentions or that you missed a step. You can then make the fix easily. This works more often than you might think. The second approach is to print and experiment.

Add print statements around where you think the error occurs. That display relevant values in the code. Run the code again and check whether the printed values match what you think they should be at that stage in the code. If something looks wrong, make a hypothesis about what the problem is and adjust your code accordingly. Then run the code again and see if the values change, repeat this as much as necessary until your code works as expected. An important part of this process is that you have to be intentional about the changes you make.

Don't just change parts of the code haphazardly have a theory for why each change might fix your problem. While you're using the print and experiment approach, you might want to pause your code in the middle of it running to see what is printed up to that point. There's a built-in function that can help with that. It's a very unusual function that's called input. Input takes a single argument, a string, and it displays that string to the interpreter, but then it pauses the code and waits.

The user is then supposed to type something into the interpreter next to that displayed string. When the user presses enter, that string is sent back to the code as the return value of the input call. In other words, input is a function that lets you get input directly from the user. For example, we can use the code shown here to ask the user for their name, then print out a reply that uses their name directly. In this example, the bold a text would be entered by the user, not the program.

Input is nice because it lets you make programs that are a bit more interactive and a bit more accessible to users. Someone can just type something into the interpreter instead of needing to change variables to hold new values. But it can also be useful for debugging. You can add a call to input in a place where you want the code to pause, and it will helpfully pause and wait for you to type something in. You can then take as much time as you want to review the printed text, then press enter once you're ready to move on.

Try using input if you're having trouble keeping track of everything that you've printed. The final approach is to trace your code. This is mainly useful when you have a really tricky bug that you can't figure out using logic. Step through your code line by line and track on paper what value should be held in each of your variables at each step of the process. Compare your traced values with what you would create step by step if you were solving the problem by hand. This might help you identify where the problem is occurring.

There are some tools that can help support you in this tracing process, but it's a good idea to practice doing it by hand first to get used to stepping through a program.

Video 12 (04:10): Debugging Worked Examples

Now, let's look at two examples of debugging code when it doesn't work the way you want it to. In this first example, we're going to look at the distance function again, to try to find the distance between two points. But something is going wrong. If I run the code, I get this error message. Take a second to pause the video and look over the code. See if you can figure out what the error is. Let's start by reading this error message. I've got a name error and it says that the name x1 is not defined.

Well, that means something's going wrong with the variable x1, but I used x1 a lot in this code. So, let's look at the line number where the error is happening. It's happening on line number six. That's the print line right here. Well, in this line I'm calling distance on x1, y1, x2, and y2 but if I look here in my code, I'm never setting these variables equal to values at this top level. I do use them as parameters in the distance function, but they aren't set to values there.

So, what I probably wanted to do is actually pass in for different numbers, maybe these four right here. And if I try running the code with four numbers instead of those four variables, now it works properly. If I really wanted to use the variables, I could have said them equal to four numbers at the top level, but I do need to have integers eventually passed to this distance call. Alright, cool. Let's look at the second example. This one's a little bit harder.

In this case, we're trying to write the function get size which takes the length of a shirt in an integer and returns the size of that shirt as a string, in this case, small, medium, or large. But again, the function doesn't work properly. If I run this, you can see that I get an assertion error. So, take a second to pause the video again, look over the code and see if you can spot the error on your own. Okay, let's take a stop at this. It's an assertion error. So, we want to see what happens when we run the function instead of what it's supposed to do.

If I run get size on 39, this is kind of interesting, I actually do get medium like the function wants but I also get 39 and that's kind of weird because the function should only be returning one value. So, let's go ahead and use rubber duck debugging to try reading through the code and seeing what's going wrong. Well, in here I've got an if statement. I'm checking if the length <= 38 and I go in here and do something. I've also got an elif and else, the elif checks if <= 40 and the else just takes all the other cases that all seems fine to me.

Now, if else's equal to 38, I'm printing small, otherwise I print medium, otherwise I print large. Actually, that seems pretty incorrect to me because I want to return small medium or large. Not print them so that I can check if get size of 39, for example, is equal to medium. So, probably what I wanted to do in here was to set up some kind of variable like size maybe and then set size equal to a value based on whatever it ought to be. So, I can set size = small, size = medium, and size = large.

If I make these changes now, I can try running the code again, but I still have an assertion error. What am I getting this time? Well, now I'm only getting one value instead of two, so that's an improvement but I'm getting 39 instead of medium. If I look at what I'm calling here, 39 seems to be the length. And if I look at my code, I'm returning the length instead of the size. So, let's try replacing that by size and I'll run the code again and there we have it. Now, it's working properly.

Now, I have to admit that this isn't very realistic debugging because I already knew what was wrong with the code before I went into solving it. Debugging on your own is often going to take more time than what I've let's here because it's going to be more authentic. You won't know exactly where to look in the first place. So, make sure to reach out for help from the program learning facilitators when you get stuck while debugging code.