Module 3: Expanding Logical Structures With Iteration

Video Transcripts

Video 1 (01:56): Introduction

Welcome back everyone. This week we'll be covering the Final Basic Building Block, Loops. This is perhaps the most exciting one among all of them. In some sense, you can say loops give you wings. You'll have full computational power to solve any computational problem we want and start building all sorts of interesting programs. Of course, learning loops is not the end of the road. We still have a lot to learn regarding how to approach complex problems, learning various programming constructs to make our lives easier, learning good programming practices, learning about computational complexity, and so on.

But in principle, after this module, we'll have all the tools to solve any computational problem, so that's exciting. Let me start by showing you the first program that I have ever written. I believe it was around middle school and I was experimenting with various things. And the output on my first program was this. To be honest with you, at the time, I was pretty proud of myself. And here is the code I wrote to produce this output. So, I guess in hindsight, I shouldn't have been so proud. This is clearly not the right way to do this.

Imagine if I wanted to create a triangle with 100 rows. Would I have to write 100 print statements? Of course, there should be a better way to do this, and indeed the way to do this would be using loops. Most programming languages, including Python, support two types of loops. They're known as the while loop and the for loop. We're going to spend about half of this module on while loops, covering the basic syntax and going over some examples. The second half will be on for loops. Let's jump into it.

Video 2 (09:31): While Loops–Part 1

As I mentioned before, there are two types of loops, the while loop and the for loop, there are good reasons to have two different types and we'll compare them later. But let's start now with the while loop. Here's the basic syntax of a while loop. The first thing you'll notice is that it looks pretty similar to an if block instead of the if keyword, we have the while keyword. We have instruction1 just before the while loop. And instruction4 comes right after the while loop. The indented code instructions two and three form the body of the loop.

This is similar to the body of an if block. But let me illustrate the difference between if blocks and while blocks by tracing through this. We start with instruction1. Then we go to the next step which is checking the while condition. Here, we check if the expression is evaluated to true or false, if it is true, we go into the body of the while loop and if it is false, we would skip to instruction4. So far, it seems to be behaving exactly like an if block but we'll see the difference now. Imagine that the expression evaluates to true.

So, we go into the while loop body, we execute instruction2 and then instruction3. Now if this was an if statement, after instruction3 we would execute instruction4. But in a while loop after executing the last instruction in the loop body, we go back up to the while condition and check the expression again. If it evaluates to true, we'll once again go into the loop body and execute instructions two and three. Let's assume that is the case. So, we execute instruction2, we execute instruction3 and then we go back up again and check the while condition one more time.

Suppose once again the expression evaluates to true. So, we go back in the loop body, we execute instructions two and three. Then we go back up again to the while condition and now suppose the expression evaluates to false. In this case we'll skip the loop body and directly jump to instruction4. At this point the execution of the loop is completed. We'll do a concrete example in a second but let me make a brief remark first.

One important thing to notice is that the loop body should change something about the expression because if the expression evaluates to true and the loop body instructions two and three do not change anything related to the expression, then the expression will always evaluate to true, which means we would be stuck in an infinite loop. There will be no way to go out of the loop and reach instruction4. So, this means the loop body has to change something related to the expression and the expression at some point should evaluate to false.

Okay, let's see a simple example now. Suppose we want to get an input from the user and we want it to be a positive integer. If the user gives us a non-positive integer then we'll keep asking them to give us a positive integer. So, while the user gives us a non-positive integer, we ask them to enter a new number. Here's how this function may look, def get_positive_int(): while(user_ input i<=0): Let user_ input = int(input("Enter a positive interger:")). And then outside the loop body we'll return user_input. Recall that the input function always returns a string.

That is why we're calling the int function to convert that string into an integer. Let's say this is our function definition. And then afterwards to test our function, let's do positive_int = get_positive_int(). And then print ("User entered", positive_ int). So, this way we'll see what the value is. Okay, would this work? There are actually a couple of bugs, pause the video here and see if you can spot them. The first bug is that we're checking if user_input is less than or equal to zero without having initialize this variable.

So, Python won't recognize this variable. So, let's run it and see the error we get, NameError. Name, user_input is not defined. Okay, to fix this we need to initialize user_input. So, let's do that. What would be an appropriate initial value? Let's see after the initialization we want the while condition to evaluate to true so that we go into the loop body. So, if we initialize it to say the value 1, that wouldn't be good because then we would skip the loop body. We can instead initialize it to a negative number or zero. Let's do 0. This should do the trick.

What is another potential bug? While the user could give us a string that does not correspond to an integer then calling the int function would give us an error. There are various ways to deal with this bug, but for now I don't want to derail us from the main discussion. So, for simplicity, let's assume that the user will always enter a numerical value as input. Okay, let's now run our

program and see what happens. Enter a positive number:. Let's enter -1 that's not positive. So, we get this prompt one more time, Enter a positive number: -5. Enter a positive number: 0.

Still not positive. Enter a positive number: 3. And now we exit the while loop. The function returns 3 and gets printed on the screen. Let's now build on this example to demonstrate another way to use while loop. Suppose after getting this positive integer, we want to print Hello on the screen that many times. How can we accomplish this? Let's define a function call it print_hello that takes one input. Let's call it num. We want this function to print "hello" num many times.

The general structure to do something some number of times would be something like this, we create a variable we can call it a counter and this will keep track of the number of iterations of the loop, we initialize it to 1, then we'll do while (counter <= num): print ("hello") Let me run print_hello(5) at this point and see what happens. It just keeps printing "hello" with no end. This is the infinite loop I was talking about earlier. If you do not interrupt the loop, this will just keep printing "hello" forever. We need to quit the program ourselves.

I set up my editor here so that pressing Ctrl C stops the program execution. So, let's stop this. The issue here is that in the loop body we're not changing anything related to the loop condition. So, it always evaluates to true. What we want to do here is increment counter by 1 at the end of the loop body. I'll go over the logic of this in a second, but let's run this and see what happens. print_hello(5) and we see "hello" is printed on the screen five times, if we do print_hello(3) we see three times.

If we do print_hello(0), nothing gets printed. Great. So, how does this work? Let's run print_hello(3) again. Notice that in the first situation of the loop until the very last line counter is going to be 1 at the end counter gets incremented. So, in the second iteration, counter is going to be 2. In the third iteration it will be 3 and so on. Now at the end of the third iteration counter becomes 4. And when we check the condition whether counter is less than or equal to num we get false so the loop is done and the function is done and that is how the function works.

We can make this function a bit more versatile. Let's change the name from print _hello to my_print. In addition to the num input we'll also give the string to be printed as input. We can call that string "s" and here in the body we'll replace the string "hello" with "s" Now we can call my_print("howdy", 4) And this will print howdy four times. That's pretty neat. In some sense, you could say we improved on the built in print function because our print function prints what you want the number of times you want it.

Let's now connect my_print with the get_positive_int(): function. Let me create a new function to bring things together. I'll call this function ask_and_print. We'll first ask the user what they would like to print on the screen. So, we do user_str = input("Enter what you would like to print: ") Then let's get a positive integer from the user. user_num = get_positive_int () Then let's call my_print with user_str and user_num as inputs. Great. Now we can see how this works. Let's call ask and_print. It asks what we would like to print.

Let me enter something here. while loops are awesome. Then it asks for a positive integer. If we enter something like -2, it asks us again, let's enter 3. And now we see while loops are awesome, gets printed three times. That's nice. We have a cool little program here that uses a couple of while loops. And we also see an example of combining multiple functions to work together.

Video 3 (08:42): While Loops–Part 2

Let's come back to the structure of While Loops. In general, when you want to repeat a block of code some number of times, this is the structure you can use. Typically, computer scientists and programmers like to initialize their variables to zero and start counting from zero. So, it would be more common to see this written as counter = 0, and then the while condition would be while counter is strictly less than num.

Now, even though we can use while loop to repeat a certain block of code some number of times, as we'll see later, it's actually better to use for loops in these kinds of situations. We'll discuss this after we finish covering while loops and for loops. Let's do another example with while loops. Suppose you want to take as input some number "n", and then you want to return the sum, 1+2+3, all the way up to n. Okay, let's think about this. Of course, we'll need a loop here. We can easily go through the numbers 1 to n, right. Let's write out that portion.

So, we have a counter set to one. We repeat , while (counter <= n): For now, let me just print the counter to see what is going on here. And then we increment the counter by one. If I run this with n being 10, then we see 1 to 10 on the screen. Good, but we don't want to print these values on the screen, we want to add them up. So, how can we do that? Pause the video here and see if we can come up with some ideas. A good idea here is to create a new variable, let's call it total, that keeps track of the sum. So, after the first iteration, total will be 1.

After the second iteration, total will be 1+2. After the third iteration, total will be 1+2+3 and so on. This way after the last iteration, total will be the sum from 1 to n. Let's now implement this idea. We initialize the total variable to zero, then in the loop body, I will increment total by counter. This way at iteration 1, total gets incremented by one, at iteration 2, it gets incremented by two, at iteration 3, it gets incremented by three, and so on. Once we're done with the loop, total will contain the sum from one to n.

Let's check it out. If we print sum to n(4), we see None. Why did that happen? That happened because we forgot to return the value total at the end of our function. So, let's add that in, and now we see we get 10, which is indeed the sum from 1-4. If we run sum to n(5), we should see 15, and indeed that is what we get, good. I should note that this kind of structure is pretty common in programming. Usually this counter variable is called an index. Think of it as an index of the iteration number.

The total variable in general you may think of it as the result that you want to return once the loop is done, let's name it result. The result gets built up at each iteration and at the end we return it. This overall template with an index variable and a result variable is quite common. But

of course how the result is built up depends on the particular problem that we're trying to solve, and the index may also have a different initial value.

For instance, suppose we wanted to modify this function, so that it takes two inputs, "m" and "n", then we want to return the sum from m to n. Then in this case, everything would be the same except that our index variable would be initialized to m. Calling now sum from m to n, three and five returns 12, because indeed 3+4+5 is 12. As another example, suppose we want to add only the odd numbers from m to n, how can we modify this code to do that? There are at least a couple of natural ways you could do this.

Pause here and see if you can come up with two different solutions. Okay, I think one natural solution would be to add an if statement in our loop body. We'll check if index is an odd number, and only then we're going to increment the result. How can we check if an index is an odd number? We can mod it by two. If index is even, mod by two will give us zero, and if it is odd, mod by two will give us one. So, if index mod two equals one, then we increment result by index. Otherwise, result will not be incremented. This will work.

The if statement is like a filter, effectively it allows us to skip some of the iterations we want to skip. How about another different solution for the same problem? Another way to skip the even index iterations, might be to increment our index by two at each step rather than by one. So, let's get rid of the if statement and increment index by two. Will this work? We have to be a bit careful. This will work if m is an odd number then index is initialized to an odd number and after every iteration, we increment it by two so index stays an odd number.

On the other hand, if m is an even number, then index will be initialized to an even number, and we will be adding up all the even numbers. That is not what we want. To fix this, we can simply check at the beginning whether m is even. If it is even we need to do a fix, we don't want to add m to our results. So, we can just imagine our starting point is m+1. So, if m is even increment it by one, and then wherein the m is odd case, which we know works properly.

Okay, let's wrap this part up with a couple of common loop bugs. It's very easy to create bugs with loops, and hopefully pointing out some of the common errors will help you debug your programs. The first bug is extremely common. I can guarantee that you'll create this type of bug in your programs. I'm going to call it off-by-one error. Let me illustrate it with an example. Here is some code for sum to n. It seems to be correct, but there's actually a subtle bug here. Can you spot it? If you analyze this closely, you'll notice that the number of iterations is one more than what we want.

In general, off-by-one error refers to a bug in which the loop body executes either one time too few, or one time too many. To spot this kind of error, what I recommend is that you check manually the first and the last iterations of the loop, and make sure that they work as intended. So, let's check the first situation here. In the first situation, counter is zero, we go into the loop body, counter is incremented by one, so counter is one, then we increment total by counter. So, we increment total by one and that seems right.

That is what we want in the first iteration. Now, let's look at the last situation. In the last iteration, counter will be equal to n, so the condition of the while loop will be satisfied and will go into the body of the loop. We increment counter by one, so it is now n+1, and then we'll increment total by n+1, but this is not what we want. we do not want to add n+1 to total. So, the last iteration here was one too many. We can easily fix this by changing the condition to counter strictly less than n.

By manually checking the first and the last iterations you can typically spot these kinds of errors pretty easily. Okay, let's move to common loop bug number two. This is something we have already discussed actually, infinite loops. When you're using while loops, it's common to run into infinite loops. And the most common way to run into an infinite loop is by forgetting to update your loop index or something related to the loop condition.

So, please always remember, in the body of the loop, you have to change something related to the loop condition, and you need to make sure that eventually that condition will be evaluating to false.

Video 4 (07:41): Problem-Solving With While Loops–Part 1

In this video, we're going to go over a couple of examples that require a bit more thought. As I mentioned earlier, once we have loops, we have everything we need to be able to compute any problem that is computable. And problems can get quite tricky. Often, you will not see the solution immediately. Therefore, it is important to have the right approach when you're trying to tackle problems. So, as we go through these examples, I want to emphasize the process of figuring out the solution rather than just presenting a solution.

Let's start by describing our first problem. Our task is to write a function named Leftmost Digit that takes as input an integer n, and returns the leftmost digit of n. For example, if n is 4092, then the returned value should be 4. Okay. So, what is the best way to approach something like this or things that are more complicated? Well, the best approach is to imagine yourself as the computer and think about how you would solve the problem yourself. Notice the change in perspective here.

You're not thinking about what instructions to give to the computer, but rather you're thinking about what you yourself would do to solve the problem. And at this stage, it is totally fine to forget about Python syntax. Think about using plain English instructions. So, think algorithmically. But as you're thinking about how you would solve the problem, it is important that you apply to yourself the same restrictions that a computer would have. For example, if I were to write down the input n on a piece of paper, it would be easy for you to immediately pick out the leftmost digit.

But the computer doesn't get to see the input number written on a paper as a sequence of digits. This number will be stored somewhere in memory. It will be stored in binary and the computer will not have direct access to the individual bits. So, it's a good idea to think of the input number

more like a black box. You don't get to directly observe all the digits or bits, but you can carry out the usual operations on numbers. You can do addition, multiplication division, and so on. So, this is the first restriction you should keep in mind.

You only have black box access to the input. The second restriction you should keep in mind is that you can only carry out basic or primitive operations. For instance, you cannot just compute for free the leftmost digit. That is not a primitive operation. If it was, we'd be done already. We need to figure out how to compute the leftmost digit by using primitive operations on numbers. One trick that helps when we think of ourselves as the computer is to imagine that the input is really large. If the input is a number, imagine a very large number.

If the input is a string, imagine a very long string. The reason for this is that if you imagine the input to be something small, like a single digit number, well, then there's nothing to do. If it's a two-digit number then you can also figure it out without much effort. These small examples don't necessarily force you to think of a general solution, but a general solution is precisely what we need.

Okay, let's now solve our leftmost digit problem. Let's imagine we have a large input. We only have black box access to it but we can carry out simple arithmetic operations. So, with these in mind, what would be a good general strategy that we can express purely in English? Not Python, just English. Now pause the video and give it a shot yourself. Okay. Here's the way I would describe the solution. Repeatedly, get rid of the rightmost digit until a single digit is left. And when a single digit is left, return that digit.

Here, getting rid of the rightmost digit is a primitive operation. It actually corresponds to doing an integer division by 10. For example, if we integer divide 495 / 10, we see that we get 49. If we integer divide 49 / 10, we get 4. So, this is exactly what we want. We want to repeatedly apply integer division by 10 until our number is less than 10. Good. We can now take our algorithm that we described in English, and turn it into Python code. Note that in the English description, we are repeatedly doing something until some stopping condition is satisfied.

When you see something like this, you should immediately think of using a while loop. The while loop is designed for situations like this. There is a condition and we repeat the loop body while the condition is true. Or equivalently, we repeat the loop body until the condition is false. With that in mind, in our code, we want to repeat until our number is less than 10. So, while (n >= 10) integer divide n // 10, and let n be this new value. Once we get out of the loop body, we know that it must be the case that n is less than 10. So, n has a single digit, and therefore, we return n.

Let's run this and see if we get correct results. Let's run it with a very big number. Okay, that seems to work. Let's run it with a small number. Good. Let's run it with a single digit. Nice. Let's try it with zero. Great. How about a negative integer? Recall that the specification given to us was that the input is an integer, which means it could potentially be negative. Now, if we run this with a small negative number, we don't seem to be getting the correct output. It actually makes sense, right?

Our while condition checks if the number is greater than or equal to 10, but a negative number would never satisfy that condition. So, we would never go into the loop body and we would just return the original input n. One easy way to fix this would be to first take the absolute value of the number and then continue with the usual code. We can use the built-in function computing the absolute value. The function name is abs, a-b-s. And with this change, our function will now work with negative numbers as well.

I want to highlight here the importance of testing our functions. If we weren't careful about it, we could have missed the fact that the original implementation was not working properly for negative numbers. To make sure our functions work as expected, we should always write test functions that cover a wide range of possible inputs. The test functions that we use have the following structure. It consists of a bunch of assert statements. Each assert statement is testing whether the function gives the expected output for a specific input.

If one of the equalities here is false, then we will get an assertion error and Python will tell us which assertion statement fails. This way, we'll know the input for which our function fails. We can then trace our code with that input to see exactly where the logic goes wrong and then fix it.

Video 5 (08:44): Problem-Solving With While Loops–Part 2

Let's solve another problem with loops. First, a definition. I'm going to call a number "awesome" if it is divisible by three or divisible by five. This is just a definition that I made up for this example. We want to write a function named 'nth_awesome', which takes a positive integer 'n' as input and returns the 'nth_awesome' number. The first awesome number is three because that is the first positive integer that is divisible by three or five. The second awesome number is five, and so on. Cool. Let's figure out how we can tackle this problem.

As in the previous example, I don't want to directly think about Python code. First, I want to imagine myself as a computer, and think about how I would solve it myself using primitive operations. Only after I have an English description of a step-by-step solution, that is only after I have an algorithm, I'm going to write Python code. A good way to construct an algorithm is to take out a piece of paper and a pencil, and then sketch out a solution. It's kind of like a picture solution, if you will. Then, we'll turn that picture solution into an algorithm.

So, you can think of the overall process as three main steps. First, sketch out a solution on paper. Second, turn that sketch into an algorithm. And third, turn that algorithm into Python code. To build a picture or a sketch solution, imagine getting some input, say, 'n = 4'. And then, figure out how to solve the problem. We want the systematic approach. So, how would you do it? What I would do is go through each positive integer one by one, and check if it is awesome or not. We'll start with one. We'll ask, "Is one awesome?"

That is, is one divisible by three or five? The answer is no. So, we'll go to the next number, two.

Is two awesome? No. Go to the next number, three. Is three awesome? Yes. At this point, we have found one awesome number. Now, you'll notice that we'll need to keep track of the number of awesome numbers we have found so far. So, let's have a variable for that, and let's call it 'num_found'. At this point, 'num_found' is one. Okay, we keep going. Is four awesome? No. Is five awesome? Yes. It is divisible by five.

We have now found two awesome numbers, so we increment 'num_found' by one. Let's keep going. Is six awesome? Yes. Increment 'num_found' by one. So, now it is three. Is seven awesome? No. Is eight awesome? No. Is nine awesome? Yes. Increment 'num_found'. So, now it is four. And this matches the input 'n', which means nine is the number we want to return. Good. This is a nice picture solution. Let's now turn this into an algorithm. So, let's write this in English as a sequence of basic instructions.

We're going to start by initializing 'num_found' to zero. Then, we're going to go through each number, 1, 2, 3, and so on. And for each number, we'll check if it is awesome or not. If it is, we'll increment 'num_found' by one. And when 'num_found' reaches the input value 'n', we'll return the corresponding number. This is now a nice English description of our picture solution. The final step is to turn this algorithm into Python code. First line is easy; def nth_awesome (n) :. The second line is also easy; we initialize a variable named 'num_found' to zero. Okay. What else?

We're going through numbers, 1, 2, 3, and so on, one by one. So, it makes sense to have a variable whose value will be one, and then two, and then three, and so on. Let's name this variable 'guess'. Let's initialize it to one. That is the first number we want to check. Now, we're going to have a loop. Under which condition will we stop the loop? Or under which condition will we want to keep iterating? While we want to keep iterating while 'num_found' is less than the input 'n'. That is the same as saying we want to stop when 'num_found' is equal to 'n'.

Good, Now, in the loop body, we want to check if the 'guess' is awesome or not. I'll write it like this: 'if (is_awesome(guess)): num_found += 1'. Here, I'm imagining that I have a function named 'is_awesome' that returns 'true' exactly when the input is an awesome number. This is a nice trick. We're not worrying about the implementation details of 'is_awesome' right now; we'll implement that later once we're done with this function. Cool. Now, after this 'if' statement, what do we want to do? We want to increment 'guess' by one, so that we move onto the next number.

And this is pretty much it for the loop. Once we go out of the loop, we want to return 'guess', which hopefully corresponds to the 'nth_awesome' number. As I mentioned before, it's very easy to make off by one errors with loops; and we may have an error here. We'll have to check. But before we can check, let's first implement the 'is_awesome' function. This is a simple one-liner: def is_awesome (num) :. To check if 'num' is divisible by three, we can check if '(num %3 == 0)'. And similarly, to check if it is divisible by five, we can check if '(num %5 == 0)'.

And that's basically it. This works because when we check, say, whether '(num %3 == 0)', then we're checking if 'num' divided by three leaves a remainder of zero. That happens precisely when three divides 'num'. Great. Let's now write some test functions to check whether our functions are working properly. Here is a test function for 'is_awesome'. The specification of

'nth_awesome' function said that the input number is a positive integer, so we'll only care about testing for positive integers.

And running our test function for 'is_awesome', we pass all the tests; so, that's great. Let's now write the test function for 'nth_awesome'. And let's run it. Okay. We see that we failed the first test case which corresponds to 'n' equals one. This is likely due to an off-by-one error. So, let's check the first and the last iterations manually when the input 'n' is equal to one. 'Num_found' is initially zero, which is less than 'n'; so, we will go into the loop body. 'Guess' is initialized to one. We check if 'guess' is awesome, but it is not.

So, we increment 'guess' by one, and repeat. Okay. That seems to be fine. Let's check that the last iteration is working properly. The last iteration will happen when 'guess' is equal to three, which is the first awesome number. In this case, 'num_found' will be incremented by one. But after that, we increment 'guess' by one, so it will be four. After we exit the loop body, we return four. So, incrementing 'guess' at the end, after we detected that it was an awesome number, messed things up.

One easy fix would be to return 'guess - 1', so that we undo that last increment by one. Let's see if that will work. And yes, it works. But if I'm honest with you, I'm not too fond of returning 'guess - 1'. I think it makes the code a little less clear. Someone seeing this for the first time would probably wonder why we are returning 'guess - 1', and they would need to figure that out. Let's see if we can come up with a nicer solution.

The problem here is that we are incrementing 'guess' after we check if it is awesome or not. And we definitely do need to increment 'guess' in the loop. But, we could increment it at the beginning of the loop body. There's no rule that says we have to do it at the end. So, let's do it at the beginning before we check whether 'guess' is awesome. This way, if 'guess' is awesome and 'num_found' reaches the input 'n', we'll return the value of 'guess' without incrementing it. This seems like a good idea, but we should now be careful about the initial value of 'guess'.

Since the first thing we do in the loop body is increment 'guess', we should set the initial value of 'guess' to be zero. Okay. I like this better. And if we run the test function, we pass all the tests. Awesome.

Video 6 (06:03): Sequences

Our next topic is going to be for-loops but before I begin introducing for-loops, I want to say a few words about sequence data types. A basic understanding of sequences will be necessary when we cover for-loops. Okay, first a reminder when you see the words data, value, or object, they all basically mean the same thing, think of them as synonyms. Similarly the words data type or just type or class basically mean the same thing. Each piece of data or value has a type.

Maybe it's an integer, maybe it's a float, maybe it's a string, types are important because the type of the data determines what kinds of operations you can use on the data. In Python, there's a category of data types called sequence types. One good example is the string data type. Another

example is the list data type. I want to very briefly go over these data types with you because we'll be using them with for-loops. And even though this will be brief, we will learn more about these data types in a future module.

Okay, let's start with strings. As we know the string data type corresponds to text values. To create a string value, we usually use quotation marks. So, if we do S = "Hello" the text Hello is assigned to the variable S. If we print(type(s)) you'll see it says <class <class 'str'> and here 'str', s-t-r really means strength. Internally a string is really a sequence of characters or a string of characters if you will. So, the string Hello is the sequence of characters H e l l, and o. Each character has an index, location in the string.

In computer science, the convention is to start indexing with zero. So, the first character has index 0, the second character has index 1 and so on. You can access the individual elements of the sequence using the square bracket notation. Inside the square brackets, you provide the index you want. So, if we print (s [0]) we get capital H. If we print s [1] we get the letter E. If we try to index into a location that does not exist, this will give you an error. For instance, printing s [5] gives us an index error.

There's a built in len function, this returns the length of the sequence. So, in the case of strings it returns the number of characters in the string. Printing the len of s gives us 5. Another sequence data type that is very useful is list. Lists are super versatile, and you'll be using them all the time. They give you the flexibility to store a bunch of values in one unit. Syntax to create a list uses square brackets. For instance, if we do L equals 2, 3, 5, 7, then this list contains four integer values. And similar to strings, each value in the list has an index.

To access the value at index two let's say we would do L [2]. A list can contain any type of value. We can define a list of integers, we can define a list of strings, we can even define a list of lists. Element access is always the same. And we can even mix different types of data in a single list. So, for example, a list can contain both integers and strings. Similar to strings, the built in len function when applied to lists gives the length of the sequence. So, it gives the number of elements in the list.

There is another very useful built in function that allows us to create some commonly used lists. This is the built in range function. If we do range n, this basically corresponds to the list 0, 1, 2 all the way up to n-1. But there's a slight technicality, you have to apply the list function to the range in order to get to the list. So, let's print list of range five and you'll see it prints the list 0, 1, 2, 3, 4. One thing you'll notice here is that the list generated does not include m, this is just a consequence of starting our indexes with zero.

So, one way to think of range and is to think of a list of length n starting at index zero. What else can we say about range? Well, it actually has a second optional parameter. If we do range m, n this corresponds to the list m, m+1, m+2 all the way up to n-1. So, think of m as the starting index and n as the ending index. But of course, we do not include m. To see this in action let's print list of range 3, 6 and we see that we have 3, 4, 5. There's a third optional parameter for range.

If we do range m, n, and k then that corresponds to the list m, m+k, m+2k, m+3k and so on all the way up to n. So, again m is the starting index, n is the ending index and k corresponds to the step. We increment the list elements by k each time. Printing list of range 2, 10, and 2 gives us the list 2, 4, 6, and 8. Cool. There's a lot we can do with lists and strings. This is really the tip of the iceberg. As I said, we'll dive into this material in more detail in the future. For now, what we briefly covered here will suffice.

Video 7 (08:28): For Loops

Most programming languages support two types of loops. We have already seen while loops, now it is time to do for loops, which is actually a much more commonly used loop. The general syntax of a for loop is the following. For variable name in sequence and then the loop body. In the loop body, you can put any block of code you want. Notice that the two important things you have to specify when defining a for loop is a variable name and a sequence. To explain what this all means, let me give a simple example.

Consider the following piece of code. 4l inList 0,1,2,3,4, and then print I. What will happen here is that in the first iteration of the loop, the variable will be assigned the first value in the sequence. So, I will be assigned zero. In the second iteration, the variable will be assigned the second value in the sequence. So, in the second iteration, I will be assigned one, and so on. The result of executing this piece of code will be to print 0,1,2,3 and 4 on the screen. If we were to change the list to 4,3,2,1,0, then we would see the numbers printed on the screen in that order.

Let's do another example, this time with strings. We know string is a sequence data type because a string is a sequence of characters. So, we would be allowed to do something like for C in the string hello print C. As before, in the first iteration, variable C will be assigned the first element in the sequence. And in this case that would be the first character, H. In the second iteration, variable C will be assigned the letter E. In the third iteration, C will be assigned the letter L, and so on. When we execute this piece of code, we see H,E,L,L and O being printed each on its own line.

The total number of iterations is five. Typically, you'll be using for loops with lists rather than strings, but keep in mind that for loops do work with any sequence data type. Let's now do another example using a list of strings. Since both lists and strings are sequences, it may not be clear how the for loop would work in this case. So, let's say we have 4S in the list Hi, Hello, Howdy. And in the body, we have print S. Okay. What will happen here? Well, the same principle applies as before. In the first iteration, variable S will be assigned the first element in the sequence.

In this case, that is the string Hi. It's not the first character in the string, it's the whole string. In the second iteration, variable S will be assigned the second string, Hello. And in the last iteration, S will be assigned Howdy. So, when we execute this code, we see Hi printed on a single line, then Hello printed on a single line, and Howdy printed on a single line. The total number of iterations is three. One of the most common ways to use a for loop is to repeat a block of code a certain number of times.

Say you want to print Hello five times. How can we do that? Well, we can do for I in 0,1,2,3,4 and then print Hello. Let's run it. Great. It works. One thing to note here is that we're not actually making use of the variable I in the loop party and that's okay. In some situations the value of the variable is not needed. Let's now think about repeating a block of code more times, say, 100 times. I wouldn't want to list out 100 numbers. We should have a shortcut for that, and there is. We already saw how to automatically create a list with the range function.

So, we can do for I in list of range of 100, and then print Hello. And when we run it, it prints out 100 Hellos. Great. That's convenient. And in fact, there's another convenience, it turns out you don't need to apply the type conversion function list. You can get rid of it. You can just do for I in range 100. And in fact, this is what you should be doing always. This will have better performance, it will actually run faster when compared to calling the list function. Okay. Let's play around with the range function with for loops.

Suppose we are given two numbers, M and N, and we want to sum all the numbers from M to N. We have seen this example when we covered while loops but let's see how we can do it with for loop. Def sum from M to N with parameters M and N, we'll need a variable to keep track of the total, we'll initialize it to zero, and at the end we'll return total. Now, in the middle, we'll have a for loop for I in range M,N+1. Why N+1? Because we call that the range from M to N+1 does not include M+1. It goes from M to N. Okay.

And in the loop body, we'll simply increment total by I. Let's quickly test this. If we print sum from M to N 3,5, we see 12, which is the correct result. How about sum from M to 0,4, prints 10, which is also correct. If we now wanted to do sum of odds from M to N, one option would be to check if I is an odd number or not in the loop body, and only increment total by I if it is odd. Another option is to first update M so that it is an odd number if it's not already, and then we make sure our range goes in steps of two. Let me do something a bit strange now.

I'll change the range so that it goes in the reverse order, starting from N go all the way down to M -1 in steps of -2. For this, we'll need to make sure N starts with an odd number. Let's run it. And this works as well. That being said, you really should not write this function like this. It is much less clear than the other ones. And clarity of your code is very important. When we write code, we're not trying to show off how smart we can be with the logic or how knowledgeable we are about the intricate details of the programming language.

We always, always strive for simplicity and clarity. We'll say more about this later, but for now, please keep this in mind when you're writing code. Okay. I would like to end this video with a brief comparison between for loops and while loops. In what situations should you be using a while loop? And in what situations should you use a for loop? Here are a couple of examples. The first example prints 0-9 on the screen. The second example is sum of odds from M to N. For both, we have a for loop implementation and a while loop implementation.

What do you think? Which implementations do you prefer? So, in these examples, the for loop is the right implementation. It produces cleaner syntax and more readable code. In general, when you have a handle on how many iterations you need, almost always, for loop is the way to go.

But there are situations where you would need a while loop because there are situations where the number of iterations is indefinite. You don't have a handle on how many times you may need to repeat. In those cases, you want to keep repeating until a certain condition is satisfied.

A good example of this is when we ask for a positive integer from the user, we don't know how many times we will have to keep asking the user for a positive integer. It may be just once or hundreds of times. We want to keep asking until we get a positive integer from the user. So, here we need a while loop. And in fact, there is no natural way to implement this with a for loop. A for loop is just not built for this type of situation.

Video 8 (08:16): Problem-Solving with For Loops–Part 1

In this video we're going to go over a more sophisticated example using for Loops. The function we want to write is called is prime. This function will determine whether a number is prime or not. Let me remind you that a number is prime if it's greater than 1, and the only divisor it has are 1 and the number itself. For example, 11 is prime, because the only divisor it has are 1 and 11. Here's a list of the first few prime numbers. Mathematicians and computer scientists love prime numbers. Prime numbers are like the atoms of numbers.

Every number has a unique prime factorization. And you can think of primes as the indivisible units of numbers, so they are very fundamental. In math for instance, if you can prove anything new about primes, you would be instantly famous, we'll give you a PhD right away. And in computer science, primes also again play a very important role. For example, for the security of many protocols used on the internet today, we want to be able to quickly generate very large prime numbers. We're not going to get into the details of this, but one of the most basic operations involving prime numbers, is to be able to check if a given number is prime or not.

And this is what we'll be implementing now. So, just to be clear, our task is the following, we want to write a function that takes an integer as input and then return is true if the integer is a prime number and false otherwise. Okay, let's solve this. What is the first thing we want to do? We're not going to immediately start typing code, right? We want to follow the same strategy that we outlined before. First, come up with a sketch of a solution. Second, translate that sketch into an algorithm, so a sequence of step by step English instructions.

And third, translate the algorithm into Python code. To come up with a sketch solution, imagine yourself as the computer and ask, how would I solve this problem myself if I was given some very large number? It's important to imagine you have a large number, we don't want to use human intuition to come up with an answer. For instance, if you imagine the input is 5, you can immediately say, I know 5 is prime. Or if the input is 10, you can immediately see, 10 is 2 times 5, so it's not prime.

With simple inputs like these, it is hard to see the mental process that goes into figuring out the answer. But if you imagine a large input say, 251 or even larger, then that will force you to really think about a systematic process to figure out the answer. Okay, as you think about solving this

problem, you can imagine that you have a paper and a pencil and that you have a calculator. The paper helps you remember information. In Python, this may correspond to using variables to store information.

The calculator allows you to carry out basic arithmetic operations. And in Python you would have access to those kinds of basic arithmetic operations. Okay, so we have paper, we have a calculator, we are given some large number, maybe 251 and we need to figure out whether it is prime or not. How should we do it? Pause here and see if we can generate some ideas. A good strategy here is really to focus on the definition of a prime number. A number is prime if its only divisors are one and itself. So, that is exactly what we want to check.

We want to go through each possible divisor one by one and see if it actually divides our number. So, if our number is 251 we want to check does 2 divide 251. If it does then 251 is not prime, if it doesn't let's continue and check if 3 divides 251, if it does then 251 is not prime. Otherwise continue with 4 and so on. If you try all the numbers from 2 to 250 and none of them divide 251, then we can conclude that 251 is a prime number and I'll put true. And in case you're curious 251 is indeed a prime number.

Good, so this is a nice solution sketch and we're done with step one. Next, we want to turn this into an algorithm. So, let's write our sketch solution as a list of English instructions. Nobody knows the input number, go through each number from 2 to num -1 one by one. For each of these numbers, check if it divides num. If you find the divisor I will put not prime. Otherwise, if none of the numbers divide num I will put prime. Okay, this looks pretty good.

Let's now convert it into Python code. Notice that the algorithm has the phrase for each, so this suggests that we should have a loop and in particular a for loop. There is also the word if, so we'll likely have an if statement as well. Cool, let's write our Python function, def is prime num. Now, the first thing we do in the algorithm is that we check for a divisor. So, we'll start with a for loop for possible factor in range 2, num. Remember once again that this range does not include the second number num, it goes all the way to num -1, which is what we want.

Okay, now in the loop body, you want to check if possible factor divides num. In Python how can we check that? We can use the mod operator, right? If we check whether num mod possible factor = 0, then we're checking if num divided by possible factor leaves a remainder of zero. And that happens precisely when possible factor divides num. Now, if this condition is satisfied, we can immediately conclude that num is not prime, so, here we can return false. Once a return statement is reached, we're done with executing the function.

As soon as we find the first divisor we return false and finish. What if possible factor does not divide num? Then we want to continue with our loop, go to the next situation and check if the next possible factor divides num. At some point we do have to return true. So, where do we put that? Suppose we put it inside the loop, then actually the for loop becomes meaningless. We will always return false or true during the first iteration of the loop, and once we return that's it, the function is done.

So, we would only check if 2 is a factor or not, and depending on that return false or true. That is not what we want, we want to put return true after the loop, we don't want to return true until we have tried every single possible factor. If we try every single possible factor and we never find one, so we never return false, then we can safely conclude that the num is prime and return true. Nice, this looks good, but before we write our test function to see if it works correctly or not, we may want to quickly think about whether we're handling edge cases properly.

What if num is one? In that case we need to return false by definition 1 is not a prime number. We also know zero is not prime, and we know negative numbers are not prime. So, maybe the first thing we should do in our function is to check if num is <=1. If it is, we return false, otherwise we proceed as usual. Now, I think this should work properly, but we can never be sure until we write our comprehensive test function. We already have one prepared, so let's go ahead and one test is prime, and we see we passed all the tests. Great.

Video 9 (04:50): Problem-Solving with For Loops–Part 2

In the previous video, we came up with a solution to the primality testing problem, but should we be happy with our solution? It does seem to work correctly and correctness is perhaps the most important thing but there are other factors involved. It's always good to evaluate our program and think about whether we can make improvements. There could be changes we could make to make our code more readable. In this case though, I think our code is pretty readable.

Another thing to look at might be the efficiency of our code. Functions with loops can potentially run for a long time depending on the number of iterations over our loops. And if we can find a way to solve the same problem with less number of iterations that can potentially make an important difference in real world applications. We will discuss computational efficiency in much more detail in a future module but it would be nice to start thinking about it now. So, let's look at our function. How many iterations can our loop have?

Of course number of iterations depends on the input. If the input number is divisible by two, then we will detect that in the first situation and return false. On the other hand, if num is a prime number then we will go through all possible divisors from 2 to num -1. So, the number of iterations is about the same as num. That may be okay for small values of num but perhaps for large values of num, say num is a billion, this is too expensive. Let's run is_prime with a prime number that is about a billion and see what happens.

We're currently running it. It's still running. And I can tell you that we won't get any answer immediately. In fact, this may take a minute or two to complete. If the input is a prime number that is close to a trillion, we won't get a result in a reasonable amount of time, that is guaranteed. Now it could be the case that there is nothing better to do. It could be the case that whenever the input is a prime number, we just need to have as many iterations as the input number itself and we just have to deal with that. But perhaps we can be more efficient.

It is definitely something worth thinking about. So, let's ask do we really need to check all possible factors from 2 to num -1? It turns out all we need to do is check up to the square root of num. This is a very interesting fact and we'll come back to it. But now I can modify my code as follows: I define a max_factor variable and I set it to square root of num and I'll round that to the nearest integer then in my for loop rather than going all the way up to num, I'll go all the way up to max_factor. Interesting, can this work? Let's run our test function and yes, it passes all the tests.

And notice that it completed pretty much immediately. The large prime number was not a problem. The difference in efficiency here is quite significant. Think about it, if the number is a prime number that is about a million, then our first implementation of is_prime will have about a million iterations our second implementation will have the square root of that number which is about 1,000.

If the input is a prime number about a trillion, then the first implementation will have a trillion iterations, our second implementation will have the square root of that, about a million. The computer will have trouble with trillion iterations, but it will have no problem with a million iterations. Okay, this seems pretty cool but there is somewhat of a mystery here. Why is it enough to check only up to the square root of num. If you think about it, what we're saying is that if there are no factors between two and the square root of num, then the number must be prime.

Why is that the case? I will leave that as a puzzle for you to figure out. And another puzzle for you is can you find further meaningful improvements to the efficiency of is_prime? The main point I want to make for now is that there can be multiple solutions to a problem. Some solutions may not be immediately obvious and they may be significantly faster. This highlights an important facet of programming. It is not just about learning the syntax of a programming language, but it is about developing problem-solving skills so that you can try to come up with the best algorithmic approaches.