Welcome to this video on developing problem-solving strategies. In this video we will move away from learning Python to instead focus on how to take a problem, break it down, and develop a strategy for solving it with programming. When posed with a problem and tasked with solving it, it helps to follow these five steps. Throughout this video we're going to step through each of these in detail. Step number one is to define the problem. If you don't have a clear understanding of the problem it's unlikely that you'll be able to solve it.
To help you understand the problem carefully read through it until you understand what is required. Divide the problem into input, output and processing. Make a list of steps and actions that you need in order to produce the desired output. Underline words that identify the inputs and outputs. Capitalise words that identify processing actions. Ask yourself these questions. What is the desired outcome? What do I need to know first to reach this goal? What steps do I need to take? Once you're okay with step one, step two is to develop an algorithm. An algorithm is basically a detailed set of instructions that you need to perform to solve a problem. Any computer problem can be solved by executing a series of actions in a specific order. Just like a recipe, an algorithm is a set of instructions that are detailed, precise and ordered, and like a recipe it should be written in simple English. Let's do an exercise. Let's write an algorithm for getting out of bed and arriving at uni for a lecture. An algorithm for this could be these series of simple steps. Imagine, however, that these were jumbled instead. The order makes a big difference. Once we have a basic idea of the steps required we can use pseudocode to formulate a basic solution to the problem. Pseudocode is an informal language that helps develop formal algorithms. It allows you to think about the program before you attempt to write it in a programming language such as Python. Pseudocode is structured and condensed English that resembles a programming language, but without being specific enough to be recognizable as a particular programming language. Take a look at this example of pseudocode next to its equivalent in Python code. See how it's closer to being plain English than it is to the strict syntax of Python. Once you've thought through

the steps and written a solution in pseudocode you may need to revise your algorithm to check that it does indeed cover all the possible inputs and that your processing steps produce the right output. This leads to step 3, testing the algorithm for correctness. The best way to test your algorithm for correctness is to step through it by hand. Have some test cases and walk through the algorithm to see if there are any ambiguous or missing steps. Maybe some of your steps are too complicated and could be hiding mistakes. These may need to be broken down further. If you're going to be testing an algorithm without a computer you need to make sure that your test cases are simple and that you know what the correct result needs to be. Then you can follow through each step of the algorithm and if the eventual output matches your expectation the algorithm is likely correct, at least for that particular test case. Once you know that it works for expected cases consider if there are obscure or boundary cases that your algorithm cannot handle. These special cases may require special processing steps. Now that you have a working algorithm it should be easy to translate the steps into Python code. Take each step of your algorithm and implement just that portion. Test each step. It also helps to write comments in your code so that you and others can understand your logic. The last step is to test your program. Compare the output of the Python solution to the hand solution. Do they match? Is your answer providing the solution that was asked for? Let's put these five steps into practice with an example. Given a wall that has a width of 5 metres and a height of 10 metres, write a program that computes and displays the amount of paint required to paint the wall. Paint coverage is 10 square metres per litre. In step one we underline the input and outputs, or in other words, the nouns. Then we capitalise the processing actions, or the verbs. Identify the inputs. We have a wall width,
five metres, and a wall height, ten metres. We'll need those. Our output will be the quantity of paint in litres. Lastly, list our processing steps. Compute
the area of a wall, compute the amount of paint required, display the amount of paint required. That's a pretty thorough step 1. Let's move on to step 2, defining an algorithm. Computing an area is easy now that we've identified the inputs. Area

will be width multiplied by height. That's 5 times 10, which is 50 metres squared. Computing the amount of paint is also easy now that we know the total square meterage. The problem statement says that it's 10 square metres per litre of paint. That's paint coverage, to give us 5 total litres
required. We've worked all this out by hand so far and we're ready to create an algorithm. These are the more formal steps from what we've just gone through. Step 3 is to test the algorithm. As we've already done the calculations we can be confident that our algorithm is working. Let's move on to step 4 and convert this into Python. We need two variables according to our algorithm. First, width, which we assign a value of 10, and then height, which we assign a value of 5. Then we compute area, so let's make another variable named area and set it to be the result of width times height. Then we compute the quantity of paint, so let's make another variable named quantity which we assign a value of area divided by 10, the coverage of the paint. The last step is to display it to the screen which we can do with a call to the print function. Step 5 is to test the implementation if we run the program we can see that it produces the correct answer, matching the answer that we calculated earlier by hand. And that's the end of this video on problem solving strategy. Hopefully these steps will help you to think through the seemingly complex problems that you may need to solve with programming. Remember that if you cannot understand and solve a problem on paper you will not be able to solve it with code.
See you in the next video.