[SQUEAKING] [RUSTLING] [CLICKING]

ANA BELL: OK, so let's get started. Today we will be continuing talking a little bit about loops and seeing some other couple nuances of loops. And then, we'll get started on our first algorithm, the guess and check algorithm. And then, towards the end, we're going to start talking about binary numbers in advance of seeing our next algorithm.

So let's do a quick recap of what we learned last time. And then, we'll do a little coding example and then we'll move on. So we saw last lecture some looping mechanisms, right? We saw while loops and for loops. They're both a way for us to control what happens in the code. And specifically, they're a way for us to repeat certain lines in the code sort of automatically, so to speak.

So with while loops, the lines of code were repeated as long as some condition held. And with for loops, the lines of code were repeated for some sequence of values, OK? And the sequence of values was something that we decided. It was numerical. That's what we saw last lecture. Today, we're going to see that the sequence of values can actually be non-numerical as well.

So both of those loop types, I guess, had-- ended at certain times, right? So the while loop ended when the condition became false. And for loop ended when we've exhausted our sequence of values. But oftentimes, we want to write programs where we break out of the loop prematurely. We don't want the while loop condition to become false and we don't want to exhaust our entire set of values in for loop.

So in order to exit a loop before the natural end comes, we can use this thing called a break statement. And the break statement allows us to exit the loop. And the loop it exits is going to be the one that immediately surrounds the break statement. So here's a little example of a nested-- two nested while loop. So one while loop and then one nested one inside it.

The outer one is going to run whenever condition one holds. And the inner one runs whenever condition two holds, OK? Now, expression a will evaluate when both condition one and condition two hold, right? So we enter the first while loop and we enter the second while loop. Both of those conditions have to be true.

But Python, as soon as it sees this break statement, Python will immediately exit the loop that surrounds that break statement. So the loop that surrounds the break statement is the one that has condition two. The condition one loop will keep going.

So as soon as Python sees this break statement, it's going to immediately stop running the while loop. It's not even going to go back up and check the condition two, which means that expression B will actually never get evaluated. So this is terrible code. We don't want to write code like this because expression B will never be run, right? But this is just to show you the impact that a break statement would have.

And expression c will then be evaluated whenever condition one is true. Now, condition two may or may not have been true along the way. But expression c will evaluate only when condition one is true, right? Condition two would have stopped being true. And then, we're going at the same indentation level as this inner loop.

AUDIENCE: It only evaluates the one time? It only gives a one and then because the break is there?

ANA BELL: Exactly, yeah. That's a great point. So it only evaluates this expression a one time because right after it evaluates it, it sees the break. And then, we immediately exit the while loop and we're done. That is a great observation. Yeah.

And that's what this code will basically show. So here it is us doing the break statement in the same structure as on the previous slide. And what we're going to do is actually just run the Python Tutor for this code just to give you some more practice running the Python Tutor.

So this is the same code as on the previous slide. I've got a for loop that goes through some sequence of values. Can anyone tell me what are the sequence of values this for loop will loop over? 5, 7, 9, and we stop, right? Because 11 is end, but we go up to but not including n minus 1. So 5, 7, 9 are the only three values we would potentially loop over.

So hit Next. We initialize my sum to zero. So in our minds, we think about the fact that we're going to loop through-- make this loop variable be 5, 7, 9. So the first time through the loop i will be 5. We're going to add i, which is currently 5, to my sum. So 5 plus 0 makes my sum 5.

And then, we immediately see the break, right? Because if my sum equals 5 is true, so we go inside and we immediately see the break statement, this line will never get executed. So we're never going to increment my sum by 1.

So the break immediately breaks out of our loop. Now, the if statement is not a loop, right? It's a conditional. So the loop we break out of is for loop. And then, there's no other loop surrounding it, so then, the program is basically done and we print 5.

Again, bad code. We would never write code like this. But this is just to show you exactly what happens with the break statement. So there's the code block for the for loop. And this is the code block for the if statement. And the break breaks out of our loop, which is the lighter purple, not the if statement.

So let's have you write a little bit of code. And this is sort of maybe a little practice with just loops in general that we saw last lecture. There's no break, really, in this particular program here, just a little bit of practice.

So what I want you to do is to write code that, basically, has a for loop running through this range. So for i in, pick one of these. I want you to write code within that for loop that counts how many even numbers are in that range, right? So including the 0.

So for range 5, your counter should basically pick up on the fact that 0 is even, 2 is even, and 4 is even, and then that's it. So it should print 3. So here is you try it inside the Python file for today. And I've already started you off with this for i in range 5 as the first one. And I'll give you a couple of minutes to just write the code in here.

OK, would anyone like to start us off with some code? Or give me-- yeah?

AUDIENCE: [INAUDIBLE].

ANA BELL: Yep. So this line of code is going to take our i, right? So in fact, what we could do to remind ourselves what i is, and this is very helpful for quizzes as well, we can write a little comment here that says I is 0, 1, 2, 3, 4. Just so we don't have to remember this fact, we can just always look here and know what i is going to be.

And then, this line of code, absolutely correct, is going to take i and grab the remainder when i is divided by 2. And if the remainder is 0, that means that the number is even. And then, what do we do inside here?

So when this is true, when it's even, how do we keep track of whether-- or how many times this condition occurs? Yeah?

AUDIENCE: [INAUDIBLE].

ANA BELL: Yes, exactly. Should we create a variable? Yes we can. So let's call it even nums. And we'll probably want to increment it by 1, right? Because here's another number that's even. So even nums plus equals 1. And then, let's remember to initialize it right before our loop, right?

So initially, before we even start our loop, we have 0 even numbers. And then, each time through our loop, we see one that's perfectly divisible by 2. We're going to increment this little counter by one.

AUDIENCE: Not the same variable.

ANA BELL: Oh yeah, not the same variable. Thank you. Even nums. Yeah. And the mental model you should have at this point or beginning is just the fact that these three lines solve our problem. It does the automatic counting for us, right? Because i will take on 0, then 1, then 2, then 3, and then 4 automatically as the loop goes through the sequence of values. And so at the end of the loop, so sort of at the same indentation level as the for loop, all we need to do is print how many of these numbers we have.

So if we run it, it'll print 3. And if we change this to 10, it'll print probably 6 because it counts the 0. Questions about-- yeah, please.

AUDIENCE: Is there anything I should look out for in the fine print on 2 instead of 3?

ANA BELL: So if you're under counting, you may be initialize-- did you initialize even nums to something else? Or maybe this is not incrementing right or maybe the range is different?

AUDIENCE: I'll try to read them to you.

ANA BELL: Yeah. It worked. Awesome. OK, so iterating through-- using for loops to iterate through sequence of values is pretty useful. Let's take another look here at this particular program. So this program is-- this set of code, this code, and this code actually end up doing the same thing. But let's look at the top one for now.

So this is a program that takes in a string s as sort of an input, so to speak. It iterates through the numbers 0 to the length of s. For index in range len s is basically going to say for index in range 13 or however many letters this string has, right? D-E-M-O space, all those letters. There's 13 of them or something like that.

So this line of code here is going to have our index take on the value 0 through 13 representing the index in s. So the lowercase d will be at index 0, the lowercase e will be at index 1, and so on. So with this index in hand, the next bit of code, if square bracket index equal equal i will check for me if this particular character is an i, or that particular character is a u.

And every time that happens, I'm going to have this print out to the screen. There's an i or u. So inside my code here, this is the first one. And I run it. It's going to print out that sentence twice because there's only two i's or u's in here. And if it repeats, it'll print it out twice. So there's one u and one i.

But this code can actually be written a lot simpler. Notice it took me a little bit of a while to explain it. And probably, at first glance, it would take you a little bit of time to figure out what it's doing. And that's because we're actually relying on the index as kind of a middleman, right?

We're iterating for loop through the index. And then, we're indexing into that index variable to grab the particular letter. It turns out that with for loops, I told you you can iterate over any sequence of values, not just numbers. And remember that strings are actually just a sequence of characters, right? Case sensitive characters.

So in Python, we can actually write code like this. So the middle box right here. It has for loop iterating through each character in the string directly. So no longer are we looking at the index, 0 through 12, but we're looking at the letter directly. So our loop variable, which I called char, but you can call whatever you'd like, is now going to take on values that are the letters themselves one at a time.

So the first time through the loop, char will be lowercase d. The next time through the loop, char will be lowercase e. The next time, char will be lowercase m and so on. And so now we've got this sequence of values that's the letters directly.

So when we check if the letter is an i or u, all I need to do is check if that character-- char, my variable-- is equivalent to i or equivalent to u. And it's going to be the same-- and it's exactly the same code. So this is the one we had before. And this is the one I just went through, and again, it prints out that sentence twice because it's the same starting string.

So the sequence of values now is our characters directory. It's the letters directly. It's not the index itself. And it turns out there's actually a much more Pythonic way to write the code, this middle box down here. So in the bottom box, the only part that I've changed is the if statement.

And I'm using this n keyword to test whether the character that I have in hand, lowercase d, lowercase e, lowercase m, and so on is in this sequence of characters I or u.

And for this case, it's not so important, right? Because in the middle box we could do if char is equal to i or char is equal to u, which is fine. But if we wanted to test if the character is one of the digits 0 through 9, this if, or, or, or would become a really long line, right?

And so all we can do is if char is in some particular sequence of characters, Python will automatically turn that into that longer if it's this or if it's this or if it's this or if it's this and so on.

OK, so the big idea here with for loops is that, yes, we're iterating through a sequence of values, but we're not limited to just numbers. And that's the cool thing about for loops you can iterate through characters directly. And we're going to see later on, we can iterate through lists of numbers, lists of strings, and a whole bunch of other things.

So let's write a slightly more complex program. This was version 0.01 of the cheerleader robots. You see up in the corner there that I wrote the robots are not mine, but the code is. So here's a little bit of code that kind of puts together iterating through strings directly and iterating through numbers directly.

So let me show you what this program is actually doing. And then, we'll go over the code. Somebody give me some noun you're really excited about. What is that?

AUDIENCE: [INAUDIBLE].

ANA BELL: What? Never mind. Give me something else that I know. What is it? Pineapples. OK, pineapples. OK, it's going to cheer for us about pineapples, and let's say we're enthusiastic level 8 about pineapples.

All right, so this is my cheerleader program. So I typed in a word and I typed in an enthusiasm level for pineapples. And then, all it does is-- and notice the repetition, which computers are really awesome for. Give me a P, P, give me an I, I, give me an N, N, and so on, right? What does that spell? And then, it does pineapples with three exclamation marks eight times because that's how enthusiastic I am about pineapples.

All right, so let's look at the code that actually does this. Notice there's two parts to it, right? There's the part that does the spelling. And then, there's the part that does repeating the word some number of times. So these are two separate loops. The spelling is up here. This for loop here. And then, repeating some number of times is down here.

OK, so the part where we do the spelling has a for loop that iterates through the letters in the word directly, right? I'm not doing anything special with these letters. So I can just iterate through the letters directly. So for w in word, where word is the input that I grabbed from the user, w is a loop variable that's going to first be p then i then n then e then then a and so on, right?

And then, I have an if else here. And if you look carefully, the only difference between the if-- what we do inside the if and what we do inside the else is whether we type in an and then the letter, or a and then the letter, right? Because some letters make sense to say give me an a as opposed to give me a a. It just doesn't sound right in English.

The letters where it makes sense to do an are defined up here. So notice they're just defined as a really long string. And so the if statement uses that in keyword we saw on the slide, right? It says if w, so if that particular character is one of these, is in this sequence of characters, then print give me an and that particular character. And otherwise, it's not one of these letters where it makes sense to say an. So then you just print give me a and then that letter.

Here, I just rewrote these two print statements using f strings, which we talked about back in lecture 2, just to show you how you could rewrite it with f strings. But it can be done both ways.

OK, so at the end of this, we've done the spelling. And then, we have a print statement that says, what does that spell? And then, the last part is to repeat that word some n number of times, whatever the user told us. So I say that number of times in a variable called times.

And then, all I do here is I have a nice little for loop that goes through however many-- how much that number is, right? So range times means it's going to be 0 all the way up through and including times minus 1. That's a total of eight times in this particular case that it loops through.

And then, all we do is print the word with three exclamation marks. Notice that this print statement that's inside the bottom for loop is not actually doing anything with our loop variable, right? Our loop variable here is i, but we're not doing anything with it. And that's totally fine.

Because all we're using in the times in the loop in this particular case is to do this action some number of times. We don't always have to do something with that loop variable. Any questions about this code? Yeah.

AUDIENCE: Could you also have used if statements for the prints?

ANA BELL: Could you use if statements for the prints? Which prints, these ones? How so?

AUDIENCE: To evaluate the w's instead of having to concatenate. I mean, not if, f.

ANA BELL: Oh, f strings. Yeah, we could have done it like this. Yeah, so this is how it is f. And then, we do the characters themselves inside the curly brackets. Oh no, that's OK. It's OK. Yeah, there's a question. Yeah.

AUDIENCE: Can you say how that last part would work since we're not actually doing it?

ANA BELL: Yeah, so the last for loop is still going to iterate through times times, right? And the loop variable each time through the loop will be 0, then 1, then 2, then 3. We're not doing anything with the i, right? The stuff that's indented is going to get done, but we're just not using the fact that i is 0 or 1 or 2 at all. Yeah, it increments itself automatically, we're just not using it. Yeah, exactly.

And that's what I said. OK, so let's have you write a little bit of code. So let's assume you're going to be given a string of lowercase letters, right? So we're not going to bother uppercase, lowercase, just assume you're given lowercase letters. It's stored in a variable s. So as an example, s is equal to abca.

I would like you to write some code that counts how many unique letters are in this string, right? So notice a occurs twice. But the count that your code should do for this particular-- in this particular string should just be 3, right? We don't want to double count the a. So there are three unique letters in abca, they are a, b, and c.

So I do have a little hint. It involves the use of an extra variable, as these programs usually go. Try to think about having this extra variable be a string that contains everything you've seen so far. So as soon as you see a letter that you haven't seen before, add it to the string variable that you've-- marking that you've now seen this letter.

And then, use this same variable to write the rest of it. As you go through your letters, make sure that you're going to check whether you've seen it already before recounting it.

So as usual, it's in here around line 76. This is the code to do it. So I'll give you a couple minutes and then we can write it together.

OK, so let me just work through it. And this is something that I think is pretty useful in a quiz situation. It's just writing things on paper first just because it's a programming and computer science class, doesn't mean we have to start coding right away. So it's really helpful to just put some ideas down on paper.

So the way I would go about this problem is, clearly, I have to touch each character in the string s. So already, for me, that's-- I need to have a loop. So as I'm looking at each character, I'm going to keep track of it. So if it's not something I haven't seen-- so if it's something I haven't seen before, what I want to do is say, OK, I have now seen this a. So I'm going to add it to a seen variable.

And then, I'm going to increment a counter, right? I've seen it once. So count maybe equals 1 now. The next time, I look at the next letter I'm going to say, it's a b. Have I seen it before? No. Let me add it to my seen variable and increment my count. Next time, I'm going to look at the letter c. Have I seen it before? No, I'm going to add it to my seen and then I'm going to increment my count.

And then, the last time I'm going to look at this letter a I'm going to say, is it already in my seen? Yep. So I'm not going to do anything with this one, right? So when I see a letter that's already seen-- that I've already added to my seen variable, I basically do nothing in my code, right?

So the most of the work happens when I encounter something I have never seen before. So does anyone have some starter code or something we can write? We don't have to write it perfectly top to bottom. We can write pieces here and there. Yeah?

AUDIENCE: [INAUDIBLE] you can find your things up there. I wrote for char in s. And I don't think this is how I'm supposed to write it, but I said if char in s in seen for false. But like I want to say if it's not in seen.

ANA BELL: Yeah, so that's a great start. So if you want to say if it's not in seen, we can just say if char or car, or however you pronounce it, is not in seen. So that takes the inverse of true or false whatever this is. Because in seen will either be true or false. And not that will be false or true. So that's perfect. Yeah?

AUDIENCE: [INAUDIBLE] but when we use the word not, do you use the exclamation [INAUDIBLE]?

ANA BELL: Oh yeah, so we can use not when we're dealing with Booleans, right? So something that an expression that evaluates to true or false. That's when we use not. And the not equal, so the exclamation mark equal, is used with other expressions when we're testing for equality, right? Like 3 not equal 2. Or a not equal b or something like that, right? So things that could be numerical not necessarily just true and false.

OK, so if char is not in seen, so if I haven't seen it before, what do I want to do? Yep?

AUDIENCE: [INAUDIBLE].

ANA BELL: Yep. So we can append the character that we just looked at to our seen list, just as we had done incrementally here. So that takes care of adding the character one by one if we haven't seen it to our seen. Good. Anything else we want to do? Or we can even test it out like this, right?

So we can print seen each time through our loop.

So first it's a, then it's ab, then it's abc. And then, the last time it should still be abc, and it is. And the last step is to just do what the problem asks us to do, which is to print how many characters are-- how many unique characters are in this list or in this string. Yeah?

AUDIENCE: [INAUDIBLE].

ANA BELL: Yeah, we can have a counter that is initially zero before the loop. And every time we add a new thing to our seen string, we can increment our counter. And then, that takes care of the bulk of the work, right? This does all the counting, all the adding to the unique seen.

And so at the end of the loop, we have this number in hand. And then, we can just print it. So with this particular case, it's 3. If we add more a's in random spots, it's still going to be 3, right? Yeah?

AUDIENCE: [INAUDIBLE].

ANA BELL: Yeah, so now that we have some code that-- basically, that works really well, we can make improvements to it. So one improvement that's suggested is instead of keeping a counter variable, we can actually just recognize the fact that the length of our seen is just all the unique characters we've seen already, right?

Because when we double up on something, we don't re-add it. So all we can do to print out the number of unique characters is to just say, I'm going to print out the length of seen, OK? And now there's no need to increment any sort of counter. And so that still gives us 3.

Questions about this code? Does it make sense? Notice there's no else, right? We just have a nice little if, there's no else. Because there's nothing to do when we've already seen the character. So we could have else pass. And pass is just some-- it's just a keyword in Python. You see it's turned blue because it's a keyword in Python. It just means do nothing, right? So we wouldn't write this, obviously.

If we had an else case, that's what we would do. We would just do nothing. OK, other questions about the code? Is that right?

AUDIENCE: [INAUDIBLE]

ANA BELL: Sorry. Say again.

AUDIENCE: [INAUDIBLE].

ANA BELL: Why are we printing the length of seen here? So we're printing the length of seen because we see that whenever we add a unique character to this seen variable, it's one that we haven't actually seen before, right? And so the only things I'm adding to my seen are things that are new. And so even as I was going through manually here, I said, I've seen the a, I've seen the b, I've seen the c, I added them one by one.

And then, when I saw the duplicate a, I didn't add it to my here, right? And so, basically, the seen already contains all the unique characters in my list-- in my string-- original string.

OK, so quick summary of what we've seen so far before we start looking at our first algorithm. So we've seen objects, right? That's how we write Python programs. We manipulate objects by saving them to variables so the values are more easily accessible. We have expressions that evaluate to different things, integers, floats, Booleans, things like that.

We added branching as a way to control-- as a control flow mechanism to our program, right? It says, hey Python, either evaluate this set of statements or this other set of statements depending on whether this condition is true. And then, we added the last mechanism for control flow, the looping mechanism that said, either loop or repeat this code while some condition is true or loop this code for this sequence of values, OK?

So really, with that in hand, we've basically have a really nice toolbox of things that we can use to write interesting programs. That's kind of all we need. But this is not the end of the class. We're going to look at other things that will make our code neater, more readable. We can write more of it more efficiently, things like that. But really, if you want to just start writing little algorithms, this is all we need in terms of Python syntax.

So the first thing we're going to apply this knowledge to is our very first algorithm called the guess-and-check algorithm. So another word for the guess-and-check algorithm is exhaustive enumeration, OK? So the idea here is that we're given a problem. We can guess a value for a solution, OK? We'll just do a guess.

And then, we'll test whether this guess is correct. Does it solve our problem? If it does, we're done. We've found a solution to our problem. If it doesn't solve our problem, we're just going to keep making guesses until we've exhausted our set of possible guesses, right? So either we find the solution or we say we weren't able to find a solution to this problem.

It doesn't mean that one doesn't exist. It just means that with guess-and-check and exhaustively enumerating all these possible values, we were not able to find a solution. So in terms of a flow chart, the way this looks is we have an initial guess. We ask, is this guess correct? If it is, we're done. And if it's not, we're going to choose a next guess.

So let's look at finding the root of a perfect square. And that's our problem. And we're going to say either we found the root of this perfect square or we say this is not a perfect square.

So with guess-and-check we can say, well, what if we want to find whether 7 is a perfect square? If it is, what is its root? And if it's not say, that it's not a perfect square.

Well, we can make an initial guess, 6. That's not the right solution. We can make another guess, 9. That's not the right solution. We can make another guess, 2. That's not the right solution. We can make a guess 0. That's, obviously, not the right solution. We can keep guessing randomly like this, but it's not going to be very efficient, right?

What we want to do is use the power of computers, and computers work with these sort of patterns in hand, right? Remember range starting from 0 following a pattern going up to some number.

So the idea is to be systematic. And then, we can really harness the power of programming and computers being able to do things really, really quickly for us. So for that same problem, finding out whether a number x is a perfect square, let's be systematic and start with a guess of 0.

Two cases, the number we're trying to find the square root of is a perfect square. Let's say 4. We're going to start with a guess of 0, 0 squared. Solve our problem? No, increment. Does 1 squared solve our problem? No, increment. Does 2 squared solve our problem? Yes, we are done.

What if x is not a perfect square? OK, let's say, 10. Let's use the same systematic approach of guess-and-check. We're going to need to add a little bit of algebra though. Because if we don't, we're at risk of potentially doing something that will lead to an infinite loop.

So the algebra we need to add to solve our problem is to say, if we were looking at a number that's not a perfect square, we need to have to find-- we have to find a way to stop, right? We don't want to guess something that's infinite. This is guess-and-check. So we need an exhaustive set of potential solutions.

So we're going to use algebra. And we're going to say, we're going to stop as soon as our guess squared becomes bigger than x. So we're going to start guessing zero then 1, then 2, then 3 then 4. And at some point, that number that guess squared, will be bigger than x. And we know we can stop because numbers bigger than that will definitely be bigger than x.

So our first guess would be 0 squared. Obviously, less than 10. 1 squared, less than 10. 2 squared, less than 10. 3 squared, less than 10, right? That's 9. 4 squared becomes 16, and we say this is where we stop. And we have not found a square root for 10, right? So 10 is not a perfect square. Does that make sense? Is that all right?

So our exhaustive set of potential solutions is 0 through 4 because that brought us closest to 10. And at 4, we've gone over 10. And we don't need to check 5, 6, 7 because it's definitely not going to be-- those values squared will definitely be bigger than 10.

So this is the code that solves that problem. We got input from the user. So what number do you want to find whether it's a perfect square or not and what is it if it is a perfect square? We have a while loop that checks one condition, right? That's our stopping condition here.

We're going to iterate through the loop when guess squared is less than x. So on that number line, we're going to keep incrementing by 1 as long as our square is less than x. So that's this while loop here.

And what we're doing inside the loop is incrementing our guess. Guess equals guess plus 1. And then, at some point, if we haven't found a perfect square or if we have found a perfect square, this condition becomes false, right? Because this is false when we have the opposite of this less than sign.

So guess squared becomes greater than or equal to x. Now, that's two very different things, right? Guess squared greater than x means we haven't found this perfect square. But guess squared is equal to x means we have found a perfect square, right? And both of those cases trigger us to leave the while loop.

So then, right after the while loop, we need to have an if else. The if else checks for one of those two cases. So the if guess squared is equivalent to x means that we exited the while loop because we found that it was a perfect square. So like 4, for example, right? If x was 4, when we hit 2, that while loop becomes false. And we exited because 4 was a perfect square.

But the 10, for example, would fall within the else clause here, right? Because we have exited the loop because guess squared 4, 4 squared 16, was greater than 10. And so that's-- then we would print x is not a perfect square.

OK, so this works for many different values, as big as you'd like. But it doesn't work for negative values. And the reason it doesn't work for negative values is because the loop never actually enters in the first place. So for example, if we look at this whether negative 2 is a perfect square, we're going to start with guesses 0 just because that's how we implemented the algorithm, right? On the previous slide, it says guesses equal to 0 right at the top.

And so guesses 0, we say, is 0 squared less than x? No. 0 is not less than negative 2. And that while loop never even enters at all, which is fine, right? Because negative 2 does not-- or negative 4-- negative numbers are not perfect squares unless we're talking about imaginary numbers. But we're not in this particular case.

However, we might want to handle the case when the user gives us a negative number. Maybe they accidentally typed in the negative sign or something like that. So we can actually take care of that case by adding a little bit of extra code around what we already wrote. So the stuff that's boxed in red is the extra code we write. Everything else is exactly the same as two slides ago.

So the only thing we want to do when we encounter a negative number is flag it using a new variable that's either true or false. And then, at the end, we can handle that flag. So if it's true, we do something. And if it's false, we do something else, OK?

So in this particular case, we've got a negative flag initially false, which means that we're going to initially assume the user gives us a positive value, right? So a negative flag equals false.

We get input from the user. And then, we check if the user gave us a negative number. So if the x is less than 0, then we're going to change the value of this flag. Neg flag equals to true. So we're going to change the value from false to true.

And then, the rest of it is the same, right? This is all the same as what we had two slides ago. Except that, at the end, we're going to check to see if the user gave us-- actually gave us a negative number, we can check with them, did you actually mean the positive version of that number or something like that?

And so, in code, the way this looks is as follows. So if we run it and we give it 4, obviously, it tells us it's a perfect square and what its square root is. 9 works. 10, it says it's not a perfect square. And then, when we give it a negative number, square or not, it just tells us negative 4 is not a perfect square. And then, it says, just checking, did you mean 4? So it does this extra print statement when the number was negative. Yeah? Question.

AUDIENCE: I didn't quite get the [INAUDIBLE].

ANA BELL: Yeah, so I can explain that again. So the negative flag equals false is just a variable, right? I just called it neg flag. It's a variable I initialized to false just to say, hey, the number I'm going to assume is not negative. And then, we only flag it-- we only change its value to true if the number was negative.

So in fact, we could have just had a little if else here, right? We don't have this line up here. We have x is equal to int. And then, we say if x is less than 0 and neg flag equals true, else neg flag equals false. We could have done that as well.

OK, so the big idea with guess-and-check is we can't test an infinite number of values. We have to stop at some point, right? So now, we've been working with the code that looks like something on the left side, right? We've been using while loops.

But we've seen that we can actually write very efficient code using for loops as well. And in fact, the guess and check method maybe intuitively lends itself better to a for loop than a while loop, right? Because we're trying to iterate through an exhaustive set of values, right?

The numbers 0 through some number, right? And so maybe a for loop is a better way to write such a guess and check algorithm. And we're going to see how to rewrite that in a little bit. But in terms of a flow chart, the way for loop would go is we sequentially go through all the possible values. When we've exhausted all the values, we say we didn't find a solution. And otherwise, the for loop just automatically grabs for us the next value in the sequence.

So let's have you work on this for a little bit. I want you to hard code for me a number as a secret number. This is kind of what we did last lecture. So secret equals 7, 5, whatever you'd like it to be. And then, I want you to write some code that goes through all of these numbers from 1 to 10 inclusive, let's say, and prints that it found the secret number.

So if the secret number is within the range 0 through 10, print that you found the number. And otherwise, don't print anything. So if you don't find the number, print nothing. And as you're working on that, and if you finish that code, think about how you would change that code to do one thing differently. If it's not found, print that you didn't find it.

So in the first version, if you don't find it, do nothing. But in the second version, if you don't find it, tell me that you didn't find it, OK? So these codes are in this Python file. And the easier version is about line 129. And then, if you work on-- after you finish that if you're done, you can just copy that code to lines about 144 and try to modify it to the new specification. So if you don't find it, print that you didn't find it.

OK, so tell me some code for the first one. So if we find the number print we found it, and otherwise do nothing. What's better, while loop or for loop? For loop. Yeah. For, let's say, i in range. How do I get numbers 1 to 10 inclusive? 1, 11. Exactly. Good. And again, I can write a little message for myself. i is 1, 2, 3, 4, dot, dot, dot, 11.

What do I do to make the check whether this number i is my secret?

AUDIENCE: [INAUDIBLE].

ANA BELL: Yep, if i equals secret, let's say print found. OK, run it. Obviously, 4 is within that range. Obviously, 100, not in that range, right? So when we had 4, it printed found. And when we had 100, it did nothing.

OK, I'm going to copy this code and paste it down here. So let's try the version now where we just make one small change to our specification, right? Now we request the code to say, if you don't find the number within this range, print that you did not find it. What are some things we can try?

Else, OK? Print not found. OK, so 4, obviously, was found. But we also printed all these not founds. Why? Yes.

AUDIENCE: [INAUDIBLE] it's iterating through the whole range, so you could try breaking out of the [INAUDIBLE].

ANA BELL: Yeah, we printed it because it's iterating through the whole range. Every time I check an i, I'm either printing found or not found. Yeah, so we could break, I guess, when we found it, right? Break. Run it. OK, then we print not found until we find it and then we break.

So we're getting there, right? It's looking a little bit better. What else can we try? Yes. Another break. We can try another break after not found. But then, the 4 is not found. Yeah?

AUDIENCE: [INAUDIBLE].

ANA BELL: Yeah, I like the idea. Yeah, you can try to do a Boolean flag. Was that your suggestion as well? Yeah. OK, let's try to do the Boolean flag way. Let's delete the brakes. Let's go back to what we had before.

So basically, our idea is-- I think what we're trying to get at is we only want to print not found when we've gone through all the numbers in the range, right? So kind of something like this, right? I want to print the not found only once at the end of my loop, OK?

But this code doesn't work either because I'm always printing not found. No matter if I do this extra print inside here, right? Because this not found at the end here is at the same indentation level as for loop. So the suggestion from a couple of you is to actually set a flag, right?

So we can set a found flag to be originally, let's say, false, right? So before I even start my loop, let me just assume it's false. And I'm going to use this flag to trigger-- or I'm going to-- I guess I'm going to change this flag whenever I found the number, right?

So found is originally false. And the place in my code where I know I found the number is here, right? When i is equivalent to my secret. And then, I can set my found flag to be true. I only call it a flag because it flags that an event happened or not. So it's kind of a Boolean event. But it's really just a variable, right? Nothing special about the word flag. It's just a variable.

OK, so now, I think the suggestion was, now that we've set our flag to true or false depending on what happened in the code, we can say if found, or I guess, in this particular case, if not found, right? The inverse of my Boolean. Print not found.

There's no else because the else was already taken care of when we had the secret-- when we found the secret within the code. So now, we print found when it's 4. And if the number is, obviously, outside the range like 100, we print not found.

We can make a small change to it, I guess. So we don't have to print found down in there. For maybe consistency or making things even, we can just say else, print, found, or something like that. And I think that should work as well. So 100 is not found and 4 is found, right?

So now we're doing things kind of consistently. We're printing out whether we found it or not down here. And inside for loop, we're just dealing with the logic of the finding or not finding it. Any questions about this code? Does it seem all right? Does it make sense?

So I'm showcasing these Boolean flags just because they're very useful for signaling that things happened in your code, right? So when you find yourself asking, how do I how do know that this thing happened or something? Boolean flag is the answer, right? Just set it to true or false, 0 or 1, a or b, whatever you want. And then, check the value of that variable later on in the code to see if the event happened or not.

So these are the two codes that we had just written kind of side by side just to show you exactly what the difference is. So here is the code where if we don't find the number, we don't print anything, right? So it's just a for loop with an if and we say we found it.

And the one on the right is the code where we did find it-- where if we didn't find it, we printed that we didn't find it. So the only things that are added in addition to the code on the left is the stuff that's bolded, right? So I just have this flag that I initially set to false.

I set it to true when this event happened. That is, I found the number. And then, I do the check at the end to print or not print found.

AUDIENCE: Can you explain why we don't use else on [INAUDIBLE].

ANA BELL: I don't use else in the if or down here. In the if? So we don't use the else inside the if i equals secret because that if or else we'll be done every time through the loop, right? And I only print that we didn't find it one time at the end, right?

If I have an else inside for loop, it's basically asking if i is the secret number. So 0 is not the secret number, we would hit the else. 1 is not the secret number, we would hit the else. 2 is not the secret number, we hit the else. And only when I get to 7, in this case, it is the secret number, so I hit the if and so on. So it's not something I want to do every time through the loop, it's-- I put it at the end because I only need to do it once. Does that make sense?

OK, so Boolean variables are a variable that is in one of two states, right? I used here true or false. But as I mentioned, you can use 0 or 1, a or b, as long as you as the programmer remember what values you're expecting this variable to take on.

Boolean variables can be used as signals that something happened in the code, right? So this could be useful in a quiz situation. We call these Boolean flags, but again, it's just a name. It's just a variable that changes state depending on if some event happened in the code.

OK, so I'm coming back to the idea of while and for loops. And we've already seen that there are many situations where for loops are a lot easier to use than while loops, OK? So when we have for loops that iterate through a sequence of values. So the guess-and-check algorithm actually lends itself a little bit better for loops than while loops.

So here's an example of us trying to find the cube root in this particular case, not the square root of a number. And again, we're only asking if this number x is-- or in this case cube is a perfect cube, OK? So the way the code works with a for loop is we're going to iterate through all the possible values.

So we have for our guests in range some number. So we're going to check all the value 0 all the way up through cube plus 1. The reason why we did the 1 is because if the user gives us the number 1 we want to check 1 itself, right? If we didn't have cubed plus 1, if we just had cubed, we would mistakenly stop at 0 even though 1 is a perfect cube.

And then, inside for loop, we just have if guess cubed is equal to cube, then we have found our perfect cube. If we have negative numbers with cubes, it's just adding a little bit of extra code. But it's not as weird as with the square root, right? Because the cube root of a negative number is just the cube root of that positive version of that number with a negative sign in front of it.

So all we're doing with a negative number as the input is saying, I'm going to iterate through all these values in through 0 all the way up to the positive version of whatever the user gave me. So this is taking the absolute value of the number the user gave me and adding 1 to it. So just kind of like the code on the previous slide, except we're doing the absolute value of it.

We're checking if the guess cubed is equivalent to the absolute value of cubed. Exactly the same as on the previous slide, except taking the absolute value of the cube. And then, we have this extra little bit that checks if the user actually gave us a negative number. So do we need to put a negative number in front of our guess?

So if the user actually did give us a negative number, let's just take-- do minus whatever value we just found for the cube. And then, we can print the cube root of this perfect cube. OK, so again, same code as before, the only difference is absolute value of cube and adding this check to deal with negative numbers.

OK, so we can actually make this code a little bit faster because, for example, when we're taking-- checking the cube root of 27, the numbers we're checking are 0, 1, 2, 3, 4, 5, 6 in our for loop all the way up to 27, right? But we can recognize the fact that when we reach 27, fine. Let's say, 26. We can recognize the fact that when we hit 3, the guess cubed is actually 27, right?

And so in for loop, it doesn't make sense to keep checking 4, 5, 6, 7 to see if those numbers are then going to match-- or be our cube root for a,, potentially perfect cube. And so that's what this code is doing. It's going to have a little if statement in here. So again, this is the same as before.

But we're going to have a little if statement that says, if the guessed cubed is greater than or equal to-- not just equal to, but greater than or equal to. Let's break out of the loop, OK?

And so when this condition is false-- or sorry, when this condition is true, guess cube is greater than or equal to, we have exited the loop. But now, just like with the square root code with the while loop, we have to see why we exited the loop. Why did we break out of this loop prematurely?

OK, one is we exited because the guess cube was equal to the cube. Or the guess cubed was greater than the cube. And so then we have a little if else conditional here that says, if we exited because it's not equal, greater than, then it's not a perfect cube. And otherwise, we exited because it was equal to, which is the same code we had on the previous slide. Check whether the user gave us a positive or negative value, put the negative sign in front of our guess, and then print the perfect cube root.

OK, so all variations of the same sort of starter code, we're just adding little bits of functionality and making the code slightly more efficient here and there. So I have another example. And this example is probably the point in this class where you're like, aha, this is what computational thinking means.

So remember these word problems from childhood, right? You see a math problem. You have basically a system of equations. Algebraically, you could probably solve it within a minute or so. We can actually apply computation to solve problems just like these. So we don't need to do it algebraically, we can just tell the computer, here's a bunch of values I want you to try. Try them to see if they match these systems of equations. And then, print out the answer.

So here's an example. I've got Alyssa, Ben, and Cindy selling tickets to a fundraiser. Ben sells to fewer than Alyssa. Cindy sells twice as many. 10 total tickets were sold. How many did Alyssa sell?

Here's some code that could solve this problem for us. I'm basically figuring out all the possible combinations for tickets that Alyssa and Ben and Cindy could sell, right? So I've got three loops, each nested, right? So Alyssa could sell 0 or one ticket or two tickets and so on.

But for every value of a list-- so Alyssa can sell zero, one, or two tickets. For every one of those, Ben can sell zero, one, or two tickets, right? So Alyssa can sell zero, Ben can sell zero, Cindy can sell zero, Alyssa can sell zero, Ben can sell one, Cindy can sell zero, and so on. So we're basically having these three for loops that make all the possible combinations of tickets.

So here I have Alyssa Ben and Cindy trying to sell tickets to a fundraiser. And then, I have my system of equations here. So in total they sell 10 tickets. So here, total two less than twice are all Boolean variables. So a plus b plus c is equivalent to 10 is the first condition I need to hold.

b is equal to a minus 2 is the second condition I need to hold. And c is equal to-- is equivalent to 2 times a is the last condition I need to hold. Those were the conditions from the previous slides, right? And so these three Booleans, whenever they hold, total is true and two less is true and twice is true. When all these things three things hold, I have found the solution to my problem.

So inside my code, this is Alyssa, Ben, and Cindy trying to sell tickets. And the code automatically tells me they sold this many each. And what's cool about this code is we can then change something about it. And then, we can run it again and immediately it tells us what the new solution is. We don't have to do it algebraically and solve it all over again.

The problem with this code and the way I wrote it specifically is it's really slow for big numbers. If I change it to 1,000 tickets being sold by three people and then a couple other changes here, just the shear fact that I've got Alyssa iterating through 0 to 1,000 and Ben iterating through 0 to 1,000 and Cindy iterating through 0 to 1,000 takes a really long time.

And so that particular code-- I'm not even going to run it-- will take a really long time if I change the values to be 1020 and twice. But instead, we can use a mix of algebra and computation to solve the problem. We recognize we actually only need a loop through one loop, right? I only care about maybe checking Alyssa's number of tickets being 0 through potentially 1,000 tickets sold.

And then, I can use my other two equations, right? Ben and Cindy. How many did they sell with respect to Alyssa? And then, I've got my two other equations here, which will tell me how many Ben and Cindy sold with respect to Alyssa's loop, right?

And then, my last equation here is that Ben and Cindy and Melissa altogether had to sell 1,000. And so with this particular code, I'm able to find the answer to the question, which is how many tickets they sold. And again, this is really awesome because now I can make small changes to the numbers and solve the problem, basically, immediately like that. I don't need to go back and solve it algebraically as I would if I were to do math.

OK, so we can apply computation to many different problems. I hope that this is a really good showcase, this word problem of what we mean by computational thinking and the kinds of things we want you to come away from-- come away with in this class.

The last thing I wanted to talk about, and I'll just do a quick intuition, is binary numbers. And this is actually a precursor to the next algorithm we're going to see in the next lecture, an approximation algorithm. It's going to be an improvement on the guess-and-check algorithm.

So far, we've seen numbers in Python they can be integers, which are whole numbers and floats, which are real numbers. But in programming, some interesting things happen when we deal with floats. And this is going to be our motivation for talking about binary numbers and then fractions and then floats in this lecture and then in the next one. So here's an example of some code.

So I've got-- it is exactly what's in the slides. I've got an integer x. And all I'm doing in this code is I have a loop through a range 10. So that means it's going to loop through 10 times. And I'm adding 0.1 10 times. So 0.1 plus 0.1 plus 0.1 10 times.

And I'm going to print whether x, the sum, 0.1 plus 0.1 plus 0.1 is equal to 1. And just to show you I'm not pulling your leg, I'm going to run it and print whether x-- so 0.1 plus 0.1 plus 0.1 10 times is equivalent to 1. And this code prints false. Not intuitive, right? If I'm adding 0.1 10 times, I should be getting 1. But I'm not in programming.

And just to show you the actual answer we get, let's print what the value of x is and then ask whether that's the same as just multiplying 0.1 by 10. So doing the loop where we add this number 10 times gives me actually 0.9999999. Whereas, just multiplying 0.1 by 10 gives me 1, as I expect, OK?

And this is the motivation for-- or I guess the precursor to our next algorithm, the approximation algorithm, OK? So we have this thing called a floating point error. And why does it happen? And since it happens, we can't do equivalency, right? We can't use the equal equal sign between floats because we get things like this that are going to completely mess up our program when we expect something as simple as adding 0.1 to itself 10 times to be 1 and it's not, right?

And so the big idea here is we have operations on floats. Because of the way floats are actually stored in computer memory, these operations introduce a very small error, super, super small, every time you do an operation with a float. But the more you do this operation that has this tiny error, the more this error gets magnified, right? And so, then, we see surprising results like that. And so that comes about with the way that floats are actually stored in the computer.

So what we have in the computer is we work with binary, zeros and ones. But humans actually work in base 10, right? We think from 0 to 9. But the computer works in base 2, either 0 or 1. And the reason it works through 0 and 1 is because of the way that the computer hardware is built, right?

It's easy for the computer hardware to say that a magnetic spin is up or down, right? 0 or 1. It's easy for the hardware to say that it has a voltage that's low or high, right? It would be a lot harder for the computer hardware to say, hey, I have a voltage that's 0 low, high, let's say, 1, or it's 1/10 of the high or 2/10 of the high. There would be too many errors introduced.

And so it's a lot easier for the computer hardware to just be in one of these two states, 0 or 1. So that's where binary comes in. And so when we're dealing with integers, this is not a problem because we can easily convert numbers that are in base 10 to base 2 that are whole numbers, integers. The problem will come when we do floats.

So you don't need to know how to do the conversion. But it will give you an intuition for what's going to happen. So the number 1507 in base 10, so that's what we have up here, is 1,000 plus 500 plus 0 times 10 plus 7, right? In base 2, we have a similar pattern. We have some whole number multiplied by some power of 2.

Here, we had the whole number be either number 0 through 9 multiplied by some power of 10. But in base 2, we're going to have either 0 or 1 multiplied by some power of 2. And if we're trying to convert the number 1507 from base 10 to base 2, because I guess, humanly speaking, the way we'd think about it is, what is the biggest power of 2 that we can have that takes us close to but not over 1507? And that's 2 to the 10, 1,024. Because 2 to the 11 is 2000 something, and that's already too big.

And then, you ask yourself, what's the next biggest power of 2 I can add to this number, 1,024, that brings me close to but not over 1507? That's going to be 256. Notice we skipped 2 to the 9 because adding 2 to the 9 takes us over 1507. It's adding 512 to 1,024. And so we repeat this process where we're basically trying to figure out, what are the biggest powers of 2 we can add in order that makes up 1507?

And it turns out it's going to be 2 to the 10 plus 2 to the 8 plus 2 to the 7 plus 2 to the 6 plus 2 to the 5. 2 to the 4 3, and 2 are all going to be zeros. And 2 to the 1 and 2 to the 0. And the bits, 1 times 2 to the 10, 1 times 2 to the 8, is basically what gets represented here, right? These whole number portions that we multiply the powers of 10 by.

And that's how we convert from a decimal number to a binary number. But again, this is kind of a human way of converting. We can actually do it in a more systematic way, a more-- not a more imperative way, right? A recipe way. Some way that a computer can actually use to take a number and convert it to binary.

And you would never have to come up with this way. But given this way of converting to binary, you should be able to code it up. So the idea here is we're going to take a number and we're going to look at the remainder when we divide it by 2.

If it's an odd number, obviously, the remainder is 1. If it's an even number, the remainder is 0. And that remainder actually gives us the last bit, the farthest right bit. And then, we can take that number and divide it by 2 fully. And then, that gives us the remaining four digits. So you see everything else just gets shifted over.

And the way the code looks is just doing successive divisions and figuring out the remainders. So I'm just going to look at the Python Tutor real quick and then we can stop. So if we're trying to convert the number 1507 following this particular recipe, all we do is we look at the remainder when we divide the number by 2.

So this is an odd number. Obviously, the remainder is going to be 1. So we add a 1 to our binary representation. And then, we're going to keep adding what happens when we divide the remaining numbers by 2. We're going to keep adding the remainder to the front of this string here.

So if we divide the number 1507 by 2, that gives us 753. And now we ask, is 753 odd or even? It's odd, so we add another one to the front of this string-- the result string. Divide 753 by 2, it's 376. This is even. So now we add a 0 to the front of my string. So notice what happens to this string as we go step by step.

376 divided by 2 is 188. What is this even number? So we add a 0 to the front of the string. 188 divided by 2 is 94. Again, it's an even number, so we add a 0 to the front of this string. 94 divided by 2 is 47. It's odd, so we add a 1.

47 divided by 2 is 23. It's odd, so we add a 1. 23 divided by 2 is 11, so we add an odd-- so we add a 1. 11 divided by 2 is 5, so it's odd, so we add a 1. And then, 5 divided by 2 is even, we add a 0. And then, 1 is our last number, so we add a 1.

And notice this is the exact same number we had when we did it in this human, thoughtful way where we were trying to figure out the highest powers of 2 we can take to go up to but not over the number 1507. But we did this using just this very iterative, very nice loopy code. And if we wanted to do a negative number, we would just add these two boxes here. It just basically means we add a negative sign in front of it. OK. Yeah.