[SQUEAKING] [RUSTLING] [CLICKING]

ANA BELL: All right, so welcome to the first lecture of 6.100L. That's our new number. My name is Ana Bell. That's two separate names, first name Ana, last name Bell-- super confusing. But I've been a lecturer here in the EECS Department for probably almost 10 years now. And I've been doing the intro course for a while. I'm really happy to be teaching this full semester version of 6.100A.

So today what we're going to do is go over a little bit of course administrative information, and then we'll dive right into some thoughts about computers, high level how they work. And then we'll start going into some Python basics. So we're going to get coding right away. So I highly encourage you, since you're in this class, to download the lecture slides beforehand, to take notes, and run code when I do.

Some of the lectures are interactive. And we'll have breaks, so there'll be a place where you can take a break to actually do some coding. And that's important-- I call them "you try it" breaks. That's important to make sure that you're actually practicing what we are learning right at this time. The main idea for lectures is, yes, I will do some teaching, but there will also be opportunities for questions and for you guys to try some programming right on the spot.

Even if you don't finish writing a program that we start talking about, I will finish it, and we can all talk about it together. And I'll show you some pitfalls and things like that. There will be lots of opportunities to practice in this class at various degrees of granularity. And then there's also lots of opportunities that I have in the handouts to do extra practice at home and through a bunch of different resources as well.

The reason why I stress participation and practice is because part of the reason you're here is you want to learn how to program. You don't know how to program yet. And programming is actually a skill. it's like math or reading. It's something that you have to practice. You can't just watch me type in a bunch of lines of code And then when it comes time to do the quiz, you automatically know how to do it. You need to do it often, more and more so that it becomes sort of second nature.

So the three big things you'll get out of this class are knowledge of concepts, obviously-- we're going to learn some computer science ideas-- programming skill, and problem solving-- problem solving skills.

Lectures and exams basically help you with your knowledge of-- test your knowledge of concepts and help you get knowledge of concepts. Finger exercises give you the programming skills. And the problem sets help you with problem solving. Basically, if you're given an English version of-- a problem in English, how do you go from that to thinking about what computer science concepts can I apply? And then after that, how do I take those computer science concepts and actually do the programming?

So what are some topics we'll be covering? We will be, at the core of it, learning computational thinking. So in the future, when you encounter a problem, your first thought shouldn't be, How do I mathematically solve this? or, How do I brute force or manually solve this problem? How can I apply computation to help me solve this problem?

And throughout these lectures, you're going to see some examples of us applying computation to a problem you might have already seen and maybe solved mathematically, which is pretty cool. Obviously, to get that, we're going to learn the Python programming language. Once we get the basics, we're going to see how we can start to structure our code to look a little bit better so we don't just have a bunch of code dumped in a file. We're going to start to organize our code and see how we can make it neat, readable, and modular.

And then towards the-- not in this lecture but in a couple of lectures and as a theme throughout this class, we're going to look at some algorithms. They're not super complicated, but they're kind of the base algorithms for a bunch of algorithms you might see in the future if you decide to take more CS classes.

Lastly, towards the end of the class, we're going to see algorithmic complexity, which basically means we're going to start asking or trying to answer the question, how do we know the programs we write are efficient? We can write programs, but how do we know that they're fast, and how do we know that they don't take up all the memory in the computer? So things like that, comparing different algorithms that do the same thing against each other.

So if there's no questions-- again, as I said, a bunch of this information is already in the handout plus more-- we can begin. OK, so let's start by talking about knowledge.

Declarative knowledge is a statement of fact, and a lot of us probably in math and in the past have worked with declarative knowledge. But this is not how computer science, this is not how this class works. In computer science what we do is we work with imperative knowledge, which is basically a recipe, how to do something. And when we're programming, all we're doing is writing a recipe for the computer to do something. That's it.

So here's a numerical example. The first statement is a declarative statement. The square root of a number x is y such that y times y is equal to x. There are many possible values for x and y that this statement can be true, right? But if we gave that statement to a computer, it wouldn't know what to do with it. What we need to do is tell the computer how to find the square root of a number and then tell us what the square root of that number is. And so the computer then needs a recipe.

So the recipe, a really simple one for finding the square root of a number, is steps one, two, three. So what we do is-- let's say we want to find the square root of 16. We obviously know it's four, but the computer doesn't. And so we give it an initial guess. Let's say the guess is three. How do we go from there?

So the steps we follow-- step one, if 3 times 3, 9, is close enough to 16, we can stop. It's not really close enough for me. So let's keep going. Step two-- otherwise, we're going to make a new guess by averaging g, which is our original guess, 3, and x over g, which is 16 over 3. 16 was the square root we wanted to find. So our next guess is 4.17. OK, using the new guess, repeat the process until we are close enough.

So we go back to step one. That's the first part of the process. We find guess squared. 4.17 squared is 17.36. So now we say, is that close enough? Not really. It's not. It's 17. It's not really even close to 16. So let's do it again. We make a new guess by averaging 4.17 and 16 divided by 4.17. That gives us our new guess, 4.0035.

OK, next step, using the new guess, we repeat the process. So 4.0035 squared is 16.277-- .0277. Is that close enough to x? Yeah, I could be happy with this. I could stop there because we're within sort of plus/minus 1. So I'm OK with that. But if we want it to be within plus or minus 1 times 10 to the negative 6 or 7 or something like that, then we would continue the process.

So really what we had there is an algorithm. It's a sequence of steps-- step one, step two, step three. There's some sort of flow of control. We had a place where we said if the guess is close enough, then we can stop. Otherwise, we do something else. We had another flow of control where we said repeat this thing. So we're kind of not going linearly, but we're changing the flow. And then lastly is a way to stop. We don't want the algorithm to go on forever. We would like to stop at some point.

And this stopping point, I was kind of vague about it. But it could be when we were within plus or minus 1 of the actual answer. And so recipes are basically algorithms, right? My grandmother was basically teaching algorithms when she would teach me to bake a cake. She didn't call it that, but she was really.

And so even recipes have that same structure. There's a sequence of steps. There's a flow of control. Like, if you don't have egg, use egg substitute. Or repeat sticking a toothpick to see if it comes out clean every minute or something like that. And then there's a way to stop. When the toothpick comes out clean, you take it out of the oven, and you eat it. And so computers are machines that execute these algorithms. They're actually dumb. Computers are not very smart. They don't make decisions on their own.

They just follow these sequences of steps that we told them to do. Computers are good at storing lots and lots of data. We can't really do that, but computers can store gigabytes of storage, terabytes even. And computers can do operations really, really quickly, which is something we can't do. They're good at those two things, but they're not very smart. They can't make decisions unless they're told to make the decisions.

So really, the computer only does what you tell it to do. And that's one of the big ideas that I want you to come away from this lecture with. Computer only does what you tell it to do. The sequences of steps that you tell it to do, that's the only thing it follows.

So a little brief history just to make you appreciate programming, Python programming language before we actually get started with it is-- so before the 1940s, we had these things called fixed program computers. A pocket calculator as an example of that. Every button was an operation. In the little screen, you could use parentheses to put a bunch of different operations together, but there was no way to store all these operations together to later put in different inputs for that same sequence of operations. You had to input it every single-- input those sequences of operations every single time.

After the 1940s, stored programs computers came into play. And they were able to store instructions to do things as data. And there was a special program called an interpreter that executed these instructions. It knew how to follow simple sequences of steps. When the program told it to go to a different location, it did. So it was basically executing these instructions.

And the instructions that it did were arithmetic and logical, so addition, subtraction, things like that; simple tests like checking for equality between two values; and moving data, so taking this value and putting it at a different memory location.

So I just wanted to give you a really brief overview, and this is not super accurate, but it gives you a sense of how exactly things happen low level in the computer. So the computer basically has memory, where things are stored. It has an arithmetic logic unit that does operations. It knows how to add things, subtract things, multiply things, compare things. And then it has the control unit, where this program counter is set. And this is where you put a program in.

So let's see if this works. This is a program. And up here is our memory. So we have a bunch of memory locations, 3456, 3457. And at each of these memory locations, we have some values stored, prefilled. So when we first run this program, what ends up happening is that the interpreter sees the first instruction, Add, the values at 3456 and 3457 together. So it goes to these memory locations here, grabs the 3 and the 4, and sends them to the Arithmetic Logic Unit.

The ALU knows how to do the addition. So it adds 3 plus 4, 7, and sends the result back here. Now, we never told it to store that result anywhere. But the next instruction says Store the value you just got back from the ALU at this memory location, 3458. So the next step basically takes that 7 and stores it at memory location 3458. Super tedious-- all we did was add 3 plus 4.

We do that again. We add the values at 7889 and 7890. So it goes in the memory. It grabs the 5 and the 2, sends it to the ALU. The ALU calculates it as 7, brings it back, and then we store that in location 7891. And then after that, all we've done is two additions. And then the next instruction says Compare the values at memory locations 3458 and 7891. So we're going to compare the 7 with the 7.

The ALU again does this comparison and says, all right, well, 7 and 7 are equivalent. So this is true or whatever it wants to give back to the interpreter. And then the last instruction here we have is Print the result of that comparison. So we print True because they were equal. Again, super high level, but it kind of gives you an appreciation for programming languages these days. This is very tedious to write if we had to write programs in this manner.

Alan Turing a long time ago showed that you can compute anything with actually an even more basic set of primitives, not addition, subtraction. But instead, with a tape, you would actually have six primitives-- move the tape left, move the tape right, read the value at the tape, put a value on the tape, erase the value from the tape, and no operation.

And so since he showed this what the result of it actually was is down here. Anything computable in one language is computable in any other programming language. So if we had some program written in Java, that basically boils down to something super long but something that is made up of these six primitives. That means that if we boil down this program to these six primitives, we can build back up the same program in a completely different language.

And that's really powerful. That's a really cool statement. Now, we're not going to be working with those primitives. We're going to be using the Python primitives, which are more convenient, and they allow us to do a lot more things in much less time. I'm going to do a little comparison as we talk about the primitives of Python with English.

So in English, some of the primitives might be words or even we can do letters or characters. But we can say it's words. With characters, we can build up words. With words, we can build up sentences. With sentences, we can build up stories. With stories, we can build up books and things like that.

In programming languages, the primitives are numbers, sequences of characters, operators like addition, multiplication, division, checking for equality, checking that something is greater than, things like that. So once we have these primitives in a language, we can start to build up the syntax of the language.

So in English, having something like noun and noun and noun doesn't make any sense. Cat dog boy doesn't make much sense. It's not syntactically valid. But noun verb noun is syntactically valid. Similarly, in programming languages, we can have two objects kind of side by side. So here, this is a sequence of characters h and i. And this is the number 5 right beside that sequence of characters. But that doesn't make any sense, right?

What does it mean to have this sequence of characters and that number right beside it? It has no meaning in Python. Instead, what we have to do is we have to add an operator in between these two objects. So here we add a little star operator in between the sequence of characters "hi" and the number 5. And in Python, the meaning to this is I want to repeat the sequence of characters "hi," h-i, five times. So this would basically give me hi, hi, hi, hi, hi.

So once we have sentences in English and expressions that are syntactically valid, we can now talk about the static semantics of the language. So in English, saying something like "I are hungry" is syntactically correct, but it's not static-- it's not-- sorry, it doesn't have good static semantics. There's no meaning-- there is no meaning to that because the "are" is for you or plural.

Similarly, in programming languages, and this will differ depending on what programming language you use-- here, in the previous slide, we saw that you can use the star operator between the sequence of characters and the number. And that meant repeat that sequence many times. But if we use a plus operator in between the sequence of characters and a number, that doesn't have any meaning in Python. So it has a static semantic error, even though it's syntactically valid, right?

We have operator-- sorry, object operator object. So, so far, we've been able to find really nice parallels with English, the English language and the programming languages. But this is kind of where things break down, when we talk about the semantics of a language. So in English, you can have many different meanings. The chicken is ready to eat means let's eat this chicken. Or the chicken is ready to eat means the chicken wants to eat something.

Programming languages, there is no multiple meanings to a program that you write. Because the computer, the machine, the language follows the set of instructions to a T, there is no ambiguity about what it needs to do. It just follows the instructions and does what it needs to do to the end, till it reaches the-- it terminates the program.

And so programs only have one meaning, but the problem is it might not be the meaning that you intended it to have. And that's when things start to go wrong. We can have syntactic errors in our program, spelling errors and indentation errors, things like that. And those are easy to catch. Static semantic errors are 90% probably easy to catch. But the problem comes in with the semantics. The meaning that you intended this program to have might not be what it's actually doing.

And that's where most of my errors happen. And that's where I get super frustrated when I program. And that's probably where you guys will get super frustrated too because you write a program that you think is doing one thing, but instead, either it crashes right away, or it runs forever and doesn't really stop, or it terminates, but it gives you an incorrect answer. It's not what you were expecting. And we'll talk about this in a few lectures.

So when we write programs, we're basically writing sequences of definitions and commands. And we're going to write these either in a file editor or in a shell. The first, today at least, we're writing in the shell directly. And half of tomorrow, we'll write in the shell because we're not really writing any-- we're not really writing many lines of code. We're just going to be-- I'm just going to be showing you some really quick things that we can do with the Python programming language.

So hopefully you all have installed the programming environment. This is the Code Editor. So tomorrow, we'll start working in here. But for today, we're really just going to work in the shell. And even in the future, you can still type commands in the shell. I find the shell very useful if there's just something really quick that I want to check, that I don't want to write a program for and then run. It's just like a simple command that I want to check to make sure it's doing what I think it's doing before I insert it in my code editor.

So here we have this. So mine is-- I guess I'm using the white theme just because I find it easier for you guys to see. This is the file editor. And this is just a bunch of expressions or-- yeah, a bunch of code that we're going to type in today. And we're going to type it in the shell today, so the thing on the right-hand side.

OK, so what exactly do we do when we write a program? At the base of it, we are going to create objects inside our programs, and we're going to manipulate them. That's it. That's what programming is mostly about at its core. Now, when we create objects, it's important-- this is kind of something we're going to come back to again and again in a more high-level setting.

But right now what I want you to understand is that when we create an object, an object has a type. And the type that an object has tells Python the things you're allowed to do with that object. So here are two examples. The number 30, it's a number. The type we'll talk about it in a bit. The type is an integer. It's a whole number. But basically, what are the things we can do with this integer, with this number?

We can add it to another number. We can subtract it to another number. We can take it to another power. We can take some other number to this power of 30. A bunch of mathematical operations, as you would expect. So that's pretty straightforward. What about this one here, this quotation capital A, lowercase a-- lowercase n, lowercase a quotation?

So this is something we'll talk about next lecture. It's called a string. And it's a sequence of characters. The quotations tell Python it's a sequence of characters. And the characters part of it are capital A, lowercase n, and lowercase a. The kinds of things I can do with this string are not the same kinds of things I'm allowed to do with the number, right? If I tried to take Ana and divide it by the sequence of characters Bob, Python would complain very much because you can't divide a string by another string, a sequence of characters. It doesn't make sense to divide it by another sequence of characters.

Similarly, I can't take Ana to some power. I can't multiply-- I can't multiply by itself, things like that. But the kinds of things that I am allowed to do on a sequence of characters is different than the kinds of things I'm allowed to do on a number. So the things I can do with a sequence of characters is I can say, well, what's the character at the first position? What's the middle character? How long is the sequence of characters? How many characters do I have?

And so now you can see that the type of the object is actually really important. Python uses it to know the kinds of operations you're allowed to do with it. And so there's actually scalar objects, and these are Python's primitives, numbers and truth values. And there are nonscalar objects. We're not talking about these yet. We'll talk about these in a few lectures. But these have some sort of structure. So for example, a list of numbers has a structure because there's a number at the beginning of the list, there's a number at the end of the list, things like that.

But a number itself doesn't have a structure. It's just the number. So what are the types of the scalar objects? What are the types of the primitives in Python? Integers, so number 5, 0, negative 100, 1 million.

Float is another type. It represents all the real numbers, so 3.27. 2.0 is a float because it has a decimal number even though to us that just means 2. But to Python, if you put in 2.0, it says that's a type float. Negative 3.14159, things like that. Bool is a Boolean. It represents truth values. And there's only two possible values that a Boolean type has, True and False. And it has to be capital T True and capital F False.

And the last one is this NoneType type. It's literally called NoneType. And it has only one special value, None. We're not going to talk about it for a bit, but we will sometime in the future. So to figure out the type of an object when you create that object, you use the type command.

So we can say something like type parentheses. And this is a command. And inside the parentheses, you say, what do you want to find the type of? So if we do type of 7, it tells me it's an int. And if you want to do the same command again, I hit the up arrow, and it automatically puts in what I wrote previously. And then if I want to do type of 0.0, it's a float because there's a decimal point.

So this is basically what I said. So we type this in the shell. And the shell tells us what the output is. So just to reiterate, int, float, bool, and NoneType are types of objects. And there can be many different objects you can create of that type.

So if you think about it, ints and floats, we basically have an infinite number of objects we can create of those types because we can have 0, 1, 2, 3, 100, 200, 300, 1 million, and all the negatives. There's almost an infinite number of values or objects that we can create of type int and float.

But bool, there's only two, the truth values True or False. And the NoneType, there's only one, this None. So that's the type, and these are the possible values, possible objects we can create. You try it. So you can just yell out the answers. There's nothing to type unless you want to check yourself so what is the type of 1234?

AUDIENCE: Int.

ANA BELL: Int. Type of 8.99? Float. Type of 9.0? Float. Type of True? Bool. And type of False? Bool. Perfect. If you ever wonder what the type of something is, you type it in here. You guys are doing well. Type is bool. Type of lowercase t true is an error, just wanted to point that out just to reiterate the fact that capitalization matters in Python.

This is our first error, by the way, guys. Very exciting. The error is a NameError, and this is the message associated with it. You also know that it's something special in Python when you have color-coded stuff. So you see capital T True, capital F False are this dark blue here, whereas anything that's not special in Python is just black. So type is a special command. This is a float, so you see they're color coded.

OK. So once we create objects, one thing we can do with these objects is to cast them to a different type. Now, this is a little bit maybe confusing because we're not actually changing the object once we've created it. So once we create the integer 3, it's there in memory. If we cast that integer to a float version of it, we're creating a new object in memory.

We're not changing the 3. The 3 already exists. We're just getting the float version of it and storing it as a new object in memory. So when we do float 3, this is a command that gets for me the float version of the integer 3. So that will give me 3.0. So for example, this is what I had, float 3. The output is 3.0.

If I do int of 5.2, it truncates it, and it gives me the integer portion of this float. If I do int of 5.9, it still truncates it and gives me the integer version of this float. It doesn't round. I'm just asking for the integer version of this float.

Some operations like round is an operation we can do has an implicit cast in it. So if I round 5.9, it's actually going to round it to 6.0 and then cast it to an integer. So notice it doesn't give me as an output 6.0. It then rounds it to just six. So that's basically what I said in the example.

So let's have you try this. What are the types of the following? I don't need the values but the types. So if I get type of float of 123, what is the type of that? Float, yeah, exactly. Yep. What if I round 7.9? What's the type of the result? Int, yep. What if I create a float of the round of 7.2?

AUDIENCE: Float.

ANA BELL: Yes, good. Float would be 7.0. And the int of 7.2?

AUDIENCE: 7. Int.

ANA BELL: Int, yes, exactly. I want the type not the value. And the int of 7.9 is an int, exactly. Awesome, good. OK, so we've created a bunch of objects. We know that we can create a bunch of objects in our programs. What do we do with them? Well, we can combine them into expressions. So let's say we have 3 plus 2. I've got object, operator, object. Cool, syntactically valid in Python and has no static semantic error.

So if I do that in Python, it's going to be OK. 3 plus 2, 5. And the type of 3 plus 2 is an integer. So basically what I've done here, I've put an expression within this type command. And that's OK. That's, in fact, encouraged in Python. You don't just want to calculate and then stick in. That would be very, very tedious. So you can insert expressions in many, many different places.

So here we have 3 plus 2, 5 divided by 3. Again, we've got 5 divided by 3 has this decimal value. And the result has a float-- is of type float. So the important thing to remember when we're doing expressions is Python reads the expression, but it does not store the expression in memory.

What it does is it reads the expression, evaluates it to one single value, and then it stores the result value in memory. So it never stores the expression. It evaluates the expression and then stores the value. And so this is the syntax for an expression-- object, operator, object, as we just saw. And that's really-- and the idea I said before, where Python stores values of expressions, not the expressions themselves, is really, really important.

So this is my first big idea slide. I decided to insert these because I think they stress the importance of several concepts. So I hope this is one. So we're taking expressions. They can be as complex as you'd like. We can use parentheses, a bunch of-- it doesn't just have to be object, operator, object. It can be more complex than that. But basically, however complex that expression is, we evaluate it, and we replace it with one value.

And the expression can be something like this. It doesn't just have to be something that's mathematical. This was a mathematical expression, but this is also an expression. And it evaluates. So this entire thing evaluates to this word, this word which represents the type integer.

So here are some more examples. 3 plus 2, again. We've got these examples with the parentheses, 4 plus 2 times 6 minus 1 obviously gives us the number, 35. And then we can insert expressions wherever we'd like. So here I'm inserting that specific expression in the type command. And this is also an expression, like I just said. And its result is int. And similarly, we can also insert that expression here. And then we can wrap that around cast. And it gives us a float. Yes?

AUDIENCE: So when you're inserting expressions [INAUDIBLE] include the operators-- [INAUDIBLE] operators in?

ANA BELL: When you're inserting-- sorry, when you're inserting what?

AUDIENCE: Well, since you said they're expressions, and you said that you need like object, operator, object, expression, type. What would be the operators in this case?

ANA BELL: Oh, I see.

AUDIENCE: How are they defined?

ANA BELL: Yeah, that's a good-- that's a good question. So in this particular case, the type and the float are not-- there is no operator I guess in this particular case. It's more like a command that gives us an output. But there is still some-- there is still an output that it gives us. So we can then take the result of this and save it somewhere else.

Sorry, yeah, I guess the example I gave on the previous slide was just an example of an expression where we could do object, operator, object. Yeah.

Yeah, so when we have these-- I guess it works for mathematical expressions. Mathematical expressions work left to right, just like in math. Parentheses can override certain precedents. If we have commands that have computations, then we have this command with the parentheses. And we evaluate what's inside the parentheses first. So we work our way in to out in that particular case.

So here are some examples. Let's have you try these. So we can type these in our console. What are the values of the following expressions? So 13 minus 4 divided by 12 times 12. So we can try that. I don't know off the top of my head, so we'll have to type it in. 0.0625, OK. So the value of that expression is a float, right? 0.0625. What's the value of the expression type 4 times 3?

AUDIENCE: Int.

ANA BELL: Int, yeah. What about the type of the expression 4.0 times 3?

AUDIENCE: Float.

ANA BELL: Yes, exactly. That's very good. So type of 4 times 3 is int. But 4.0 times 3 is a float. Good. And then what about int of a half or of 1 over 2?

AUDIENCE: So it's 0.

ANA BELL: Yeah, exactly, it's 0. Yep, because it's 0.5, and we truncate to 0. The reason I had this here is because it leads nicely into this slide. You don't have to memorize these rules. You can always check it out in the console. But there are some rules for the resulting types when we do operations. So when we do operations with numbers, addition, subtraction, and multiplication always yield an integer if both of the operators are integers.

If one is a float or both are floats, then it gives me a float. Division is different. No matter what types you divide, you will always get a float. Now what about this // and this percent? These are actually useful operations. They kind of go hand in hand with division. So when I do 5 divided by 3, it's this 1.667.

// is basically a floor or getting the integer portion of the division. So 5//3 gives me one. It truncates the fraction. The percent gives me the remainder. So 5%3 gives me the remainder when I divide 5 by 3. So it's going to give me-- give it to me in a whole number. So that's going to be 2 because there's 2 left over when I divide 5 by 3.

So these are pretty useful operations, the // and the percent, when we do mathematical programs. The last thing is the ** is how we denote power, exponentiation, kind of different than you might be used to in math. So 2 to the power of 3, 8. 2 to the power of 3.0, 8.0. And the rules for integer division, percent, and exponentiation are just like addition, subtraction, multiplication. If one is a float, then the result will be a float as well.

Yeah. OK, and we talked about the type of output. So I think I briefly mentioned this. The operator precedence is exponentiation and then multiplication, division, percent or remainder at the next level, and then addition, subtraction at the bottom. But you can always override these using parentheses.

OK, questions so far before we move on? Yes.

AUDIENCE: So why does division-- why does it always result in float if you have 9 by 3 and that's [INAUDIBLE] why does it [INAUDIBLE]?

ANA BELL: Yeah, so the question is, why does it always result in a float? If it didn't, I think it would the operation itself would have to do extra work to figure out whether it's a whole number or not. So I think it's just easier that it gives us always a float, I guess. Previous versions of Python, the / was actually, I think, integer division, which is super counterintuitive because you would use that in your program. And then you would basically integer divide, and things would go wrong. But again, just a design choice on behalf of the programmers. Other questions so far?

OK, so we have a lot of objects. Objects have different types, again, floats, integers, Booleans. What can we do with them? So far, they're kind of just sitting in there, and we can get properties about them. But what we'd like to do is write programs, basically trying to automate some things about these objects, manipulate them to help us achieve a more complicated and interesting program.

So what we can do to get to that end is to start assigning names to some of these objects. If I create an object for pi in my program to 20 decimal places somehow, and I have that number in my program, that float in my program-- if I want to use that number in many different places in my program, I'd have to copy and paste it a whole bunch of times so far, which is very tedious, lots of errors will happen. I don't want to do that.

So instead what I can do is I can give a name to this ridiculously long value of pi called pi. And then I can just use this name anywhere I want to grab that ridiculously long value for pi in my program. It's a lot easier to read. It's a lot easier for me to write this program. And it leads to a really nice and neat program.

So what we can do is we can start saying that the float 0.001 will be referenced by the name "small" or the 100.4 will be referenced by the name "temp." So what we want to do is create these things called variables. And a variable is different in computer science from a mathematical variable or variables that you've known so far in math.

So math variables come back to the idea of declarative knowledge, a declarative statement. You can have something like a plus b is equal to b minus 1 in math, or x is equal to-- or x times x is equal to y, and that's perfectly OK. In math, we basically say that variable x represents all the square roots of y. That's not going to fly in computer science.

In computer science, we don't have-- we don't do declarative knowledge. We do imperative knowledge. And so what we're working with in computer science is a bunch of assignment statements. So what we can do in computer science is we're going to basically bind a value to a variable. So we're going to say this variable name is bound to this value. Every time I want to grab this value, I'm going to invoke this variable name. So here are some examples.

I've got a is equal to b plus 1. The thing on the right-hand side will evaluate to some value as long as I have something that b has a value for. I've got here m is equal to 10. So m is a variable. Its value is 10. I've got F is equal to m times 9.98. So again, I have an expression on the right-hand side, and that's OK. I'm going to use the value of 10, so F's value will be 99.8. Yeah.

AUDIENCE: Can you put it so that for F-- is it like this one value of m? Or can you have it so it's going to be whatever m assigned recently?

ANA BELL: Yeah. The question is, can you have m whatever it recently is? So in this particular case, I just have these two lines. And m will be whatever 10 is. But we'll see in a couple lectures that we can write a loop where you change m. And then every time you change m, you immediately calculate F. And then it'll calculate F based on the new value of m. But if we just have these two lines, that's all there is. It just uses 10. Was there another question?

So in computer science, you have only one variable to the left of this equal sign, called the assignment operator. And you have a value to the right-hand side of the equal sign, the assignment operator. So one variable basically maps to or binds to one value.

So the equal sign is an assignment statement. It's not equality. It's not a solve for x type of situation. It's just an assignment. It binds this name to this value. So the way that we figure out the name with the value is, well, if we have this assignment statement here, we first look at the right-hand side. So we always start with the right-hand side. And we evaluate it.

Remember, we have an expression on the right. We have to evaluate it to one value. So this will be 3.14, whatever it is, 1.159. And then we take that value and bind it to the name pi. So anytime I type in p-i, "pi," in my program from now on, Python will automatically grab 3.14159 from memory. So it's bound to that value now.

OK, there are some rules. Did I have them on the previous one? Yes, there are some rules to variable names, but we'll talk about that in a bit. For now, I want you to tell me if any of the following are allowed. If I do x is equal to 6, is that allowed in Python?

AUDIENCE: Yes.

ANA BELL: Yes, it is. Good. Because I have one variable name bound to one value, 6. What about 6 equals x? It's just backward.

AUDIENCE: No.

ANA BELL: OK, good. 6 equals x is bad, syntax error. How about x times y equals 3 plus 4?

AUDIENCE: No.

ANA BELL: No, exactly, because the thing on the left has an operator in it. And operators are special. So it can't have-- you can't have a variable with that * as a name. How about xy equals 3 plus 4?

AUDIENCE: Yes.

ANA BELL: Allowed, yes, exactly. I was hoping to get you guys with that, but I didn't. Xy equals 3 plus 4 is OK. There was no error. And then I can invoke the name of the variable I just created simply by typing it in. So if I type in xy, it gives me 7. And then I can do operations with it, xy plus 1 is 8. Yeah.

AUDIENCE: Before you were putting the strings with apostrophes. So wouldn't you need that?

ANA BELL: So those are strings, right, sequences of characters. Here, these are variables. So these are names that I am giving as a variable. Yeah, that's a great question. So this is going to be a string. And you notice it changed color. It has some meaning in Python. But xy is a variable that I create.

OK, so why do we want to give names to variables? Because as I showed you with the pi example, it's a lot easier to write readable code if you have variable names within your programs. So when you grab-- when you write programs, it's important to choose variable names wisely. You don't want to use just single letters. You don't want to name it something that doesn't have something to do with the program you're writing, because you're going to want to reread these programs sometime in the future. Or others might want to read your programs sometime in the future.

So here's an example of a nice program. It's just basically four assignment statements that do some calculations. The first line of the program is not really a line. It's called a comment. You can have as many of these as you like. They start with a hash. It's a line that starts with a hash. And it's basically a text that you write that helps you or others figure out what the code is supposed to do.

And usually we comment large chunks of code at a time, not line by line. Then we have these four assignment statements. So here I'm defining variable named pi bound to the value here, so not the division but 3.14159. Variable named radius bound to this float 2.2. And then I have a variable named area which is bound to the result of this expression.

So when Python sees my pi and my radius, it grabs them from memory, replaces them with the values, evaluates the expression, grabs that one value that we evaluated to 15-point-something, whatever this is, and binds the 15-point-something to the name area. Same with circumference.

Code style is something that we're actually going to look at in your problem sets. So I just wanted to quickly talk about that. Here is a program that has really bad style. Actually, that shouldn't be meh. It should be terrible or something like that. But in case you haven't noticed, it's the same program as on the previous slide. But if I gave you this program straight off the bat, you probably wouldn't know what it's doing. It's reusing 355 over 113 twice here. It's using just a and c as variable names. Its description is "do calculations." So pretty bad.

This is a little bit better. I've recognized that 355 over 113 is being used twice. So I'm saving it as a variable. But my variables are still single characters. And my comments are pretty bad. I'm basically saying what the code is doing. Please don't do that. We can see that a equals p times r times r. I see that I'm multiplying p with r squared. I don't need to read that in English.

What I would like to see is a comment like this. Here I'm commenting a chunk of code. And someone who doesn't want to read this chunk of code just reads the comment, and I already know that I'm calculating the area and circumference using an approximation for pi. That's a pretty nice comment there and good descriptive names and all that.

So we can actually-- once we create an object, a variable-- sorry, once we create an object and bind it to a variable, we can change the bindings. So we can take that variable name and bind it to a completely different value. This might not be useful right now, but it will be useful when we introduce control flow in our programs.

So to rebind a variable what that means is we're going to take the name, we're going to lose the binding to the previous value, and we're going to rebind it to a new value. So I'm going to show you how this looks like in memory. I'm going to use this sort of cloud picture to represent what happens behind the scenes whenever we write programs. And it's like a little animation to help you understand line by line what's going on.

So here we have pi equals 3.14. So the green 3.14 is my value in memory. Cloud is memory. That's my value in memory. And it's bound to this name pi. So this is my variable name. The next line, radius equals 2.2, same thing. I've got 2.2 as my value in memory, my object. And radius is the name for that object. Area equals pi times radius squared. So what happens behind the scenes is it calculates this value. It doesn't store the expression. It stores the value resulting from the calculation, and then it saves it-- or binds it to the name area.

OK, everything OK so far? We've seen this code before. Cool. So now what happens when we do this, radius equals radius plus 1? In math, that would say 0 equals 1. But we're not in math here. We're in computer science, and this is perfectly valid. We're following the rule when we have an assignment that says look at the right-hand side first and evaluate it and then bind it to the left-hand side.

So if we look at the right-hand side first, we see radius. Well, what's the value? 2.2. We see add 1 to it, 3.2. Save that in memory. And then we see the assignment. Now save it with the name radius. OK, so we can only have one variable assigned to one value at a time. This is not math. This is computer science. So you can only have radius point to one thing at a time.

With this line of code, radius equals radius plus 1. We've lost the binding to 2.2, this object in memory, and we've rebound it to the value 3.2. And that's perfectly fine. 2.2 is now just sitting in memory. We can't get back to it unless we say maybe radius equals 2.2. It just sits in memory and then might be collected later on by-- or reclaimed by garbage collection or something like that. But for now, we can't get back to it.

Now, what's the value for area at the end of these lines? Well, according to this, it's 15.1976. So it's using the old 2.2 value for radius. And that's OK because the program never told-- never had a line that said recalculate area after we changed the radius. It's just following, dumb, line by line. It doesn't know that, hey, if I change the radius, the user might want the area changed. It doesn't make those connections. It's just following instructions.

And that's OK. If we want it to change the area, we would have to copy this line and paste it after we've changed the radius. And then the area would change as well. Does that make sense? That's kind of an important part of this lecture. OK, cool.

So big idea here is our lines are evaluated one after the other. We're not skipping. We're not repeating things. That's something we're going to learn about later. But for now, line by line. So here's a little you try it. These three lines are executed in order. What are the values for meters and feet variables at each line? So how about at the first line, what's the value for meters after we execute the first line? 100. What about feet?

So at the end of the first line, there is no value for feet yet. How about after the second line? 328.08. Right? How about the value for meters?

AUDIENCE: 100.

ANA BELL: 100 still. And what about after the third line? I'm changing meters to 200. Exactly, yeah. Meters is 200, but feet is still 328.08. And this is something I want to show you guys today. And we're going to use this Python Tutor a lot more in the future. Python Tutor is a nice website that allows you to step in your code-- step through your code step by step.

So at each line that you execute, you get to see the values of all the variables in the code. It's a very useful debugging tool. I hope you'll try it out today and on Monday, maybe, for the finger exercises if you're having trouble. And you can use it for quizzes to help you debug. But I can just show you. It's pretty simple here because it's just a step by step.

So we step through. So the red says the line I'm going to execute. Green is the line I just executed. So I just executed meters 100. So here I have my meters variable with the value 100. Step through next. So I just executed feet equals this. So I now have a variable named feet with a value 328.08. Meters still 100. And then meters 200, feet remained 328.08.

So obviously, this is a pretty simple program to run the Python Tutor on, but you can imagine using it in more complex settings. How about one more? And this is my last example. I want you to try to write a program that swaps the values of x and y.

So originally-- and I'll draw this, the memory diagram real quick. So we have-- this is our memory. We have x is bound to 1. Y is bound to 2. And what I want to do without saying x equals 2, y equals 1, what I want to do is swap the values. I want x to be associated with 2 and y to be associated with 1 but only using commands like this.

And so the code here is buggy. That means it's wrong. It has an error in it. Well, let's step through-- let's step through a little bit at a time. Y equals x. What do I do when y equals x here? Yeah, exactly, y is going to move from 2 to 1. Now, what happens when I do x equals y? Yes, x stays the same.

My first line, y equals x, lost the binding to 2. And now it's all messed up because I can't get it back. So instead-- so if you didn't understand this, you can click Python Tutor and just step through step by step on your own. But how can we fix this?

AUDIENCE: Create a third variable.

ANA BELL: Create a third variable? Yeah, that's a great idea. Yeah, we can create a third variable. So x is 1, y is 1-- y is 2. So we can create a third variable. What do you want to make the variable equal to? X or y? Yeah, either one. I made it y, so let's do y. So here I've got a temporary variable called "temp," and I made it equal to 2. And now what can I do? Which one can I reassign now? X equals y, or y equals x?

Exactly, y equals-- if I do x equals y, I lose my binding to 1, and it messed up again. So y equals x is OK to do. So I'm going to lose the binding from y from 2 and bind it up to 1. And now what do I do? Yeah, now I can safely reassign x to temp. So I can say x is equal to temp because temp points to 2. And I want to make x point to 2 as well.

So in terms of code, that's sort of the diagram. But we can write the code. So you don't-- let's see. We don't write it in here, but on your own, you can write it in here if you'd like. Or we can do it together. So x is equal-- oops. X equals 1, y equals 2. And then we had temp. We wanted to assign it to whatever y was. So we say temp is equal to y.

And if you want to check the values of the variables, you can just invoke the names. So x is 1, y is 2, and temp should be whatever y is, 2. OK, good so far. So now I'm at the step here, I think, right? I've just created this. And then the last thing I need to do is lose the binding from x to whatever temp is.

So I want to do this operation here, which means I want to assign x to be equal to temp. So now x is 2, y is 1. What did I do? Yeah, so this happens sometimes. We can just start all over. So y equals temp. Sorry. Temp equals y. Y equals x.

Y is one. X is 1. And then x equals temp. Y is one, x is 2. So it's OK if things go wrong. They will go wrong. We can just start all over in this particular case by redefining our variables and just trying it out all over again. So that's kind of what the shell is for. That's what I use it for. That's what we're going to use it for in the future, just to do quick things like this and also things like checking the types and other commands we've done earlier.

OK, so any questions before we do the summary? Was this all right pace or was it too fast? Or it was OK? OK, good. Thumbs up is good. So let's do a quick summary. We saw that we can create programs by manipulating objects. We created objects in Python. And we saw that objects have a particular type. The type that the object has tells Python the things that you can do with that object.

We can combine objects in expressions. And these expressions evaluate or boil down to one particular value. Objects or values can be stored in variables. And these variables allow us to access these values with nicer names later on in our program. And then we're able to write neater, more legible programs as well.

So the equal sign-- I showed you a couple of differences between math and computer science. The equal sign was one notable difference. The equal sign in math is declarative, and the equal sign in computer science is an assignment. You're basically saying this is associated with this. And we're not doing any sort of equality in computer science.

And yes, computers do what you tell them to do. That's kind of the big thing here. Line by line, it executes starting from the top, goes line by line. So far, we haven't seen any places where the computer makes a decision. But next lecture, we will see how we can insert decision points in our programs for the computer to either execute one set of code or another set of code.

All right, so that's the end of today's lecture. Thank you all for joining. I will see you on Monday.