PROFESSOR: Hi. Today I'd like to talk to you about some notable aspects of Python that you'll encounter in the 6.01 software, as well as in the general sense when working with Python. A lot of these little tidbits of Python have some interesting history associated with them, especially related to the history of computer science. And also, I want to indicate some things that tend to mess up, especially, first time programmers, but also especially people that are new to Python.

First, I'd like to bring us back to last week. We were talking about object-oriented programming and inheritance. Object-oriented programming is a programming paradigm. It's in the same category as imperative programming or functional programming. And you might say I have a good sense of the fact that in object-oriented programming, everything is an object. But I don't really have a good sense of what constitutes imperative programming or functional programming.

I'm going to focus on functional programming right now, because it has more of a historic root in the academic development of computer science. Functional programming is like object-oriented programming in that everything is a function. It refers to the idea that you want to write as much of your code as possible in a purely functional manner. And in particular, you're looking for the ability to avoid things like side-effects or enable many different kinds of evaluation.

Those kinds of things are not the subject of this course, but I think they're worth noting to figure out why we're bothering learning about things like lambdas and list comprehensions in the first place. Speaking of which, the first thing we need to talk about before we even talk about things like lambdas and list comprehensions, is the concept that in Python, everything is an object. But because functions are first-class objects, we can also treat Python to a large extent like a functional programming language.

What do I mean when I say functions are first class objects? I mean that a function can be the return value of another function. I mean that the function can be passed

in as an argument to a function. And I mean that functions can be assigned variable names and manipulated the same way that we manipulate any other piece of data structure. This is important, because if you want higher order functions or functions that actually modify or use other functions as a part of you know, whatever it is they do, then you need to be able to interact with the function like it's any other object, in part pass it in or return it out.

Let's look at an example. So let's say I start off with a very basic function, right? If you want to square some sort of value, probably numeric in this case, then all you have to do is multiply it by itself. Pretty simple. Good place to start.

As a consequence of the idea that functions are first class objects, I can write a function that takes in a function as an argument and then develops a return function that uses the function that I passed in, on itself. So any arguments that I would pass to someFunction, I would pass into someFunction, take the return value of this function call, and then pass that into someFunction again. That's what returnFunction does. And down here, I return returnFunction.

Note that the object type of this return value is a function. So once I have returnFunction, I can actually pass it arguments, have them run through someFunction, not once but twice, and then get out a value that's of the return type of someFunction.

Note that I made some assumptions while writing this function. First, I've assumed that someFunction returns out the same number of arguments, and either accept an arbitrary number of arguments or accepts the same number of arguments as it puts out. The other assumption that I've made is that the data type that someFunction returns is the same as the data types that someFunction accepts. Or the things that someFunction does to its arguments can be done to multiple kinds of arguments.

But that's a whole different argument. Right now, we're just focusing on the fact that we pass in some arbitrary number of arguments, call someFunction on it, call someFunction again on the return value of this, and return that as something you

can do to whatever it is, someFunction operates on. OK. Let's review an example, because I promise it'll be more clear.

Let's say f is going to be the composition, a square on itself. If I do that, I end up with a function that operates on an arbitrary number of arguments-- that's not true. I end up with a copy of square that takes in the same number of arguments as square called on itself. So here a square is substituted for someFunction. Here a square is called on that call of square. And here is what f is assigned to.

So when I call f of 2, I'm going to substitute 2 in for args. I'm going to call square on args. If I call square on args, I get out 4. And if I call square on args again, or if I call square of args where args is defined as the return value of this, then I'm going to call square on 4, I'll square 4, and I'll get 16.

Take a second to type it into IDLE, and make it make sense to yourself. All of this code should compile. Mess around with the parameters, if you're having trouble convincing yourself that it does.

Ok. Now I think we're ready to talk about lambdas. If you can pass in functions as arguments or return them as values or assign them to variable names and just treat them like any other data type, then you should be able to treat them as some sort of raw value. And that's where lambdas come in.

Lambda is a key word in Python that tells you you're about to use a function that you have not given any sort of name or defined in any place in your code or environment beforehand. You're just going to start talking about something that you would like to do to a given number of arguments, and then what you want to return out.

Lambda has roots in lambda calculus which, if you're familiar with the history of computer science, you've probably heard of. Now might be a good time to look up lambda calculus, if you've never heard it before or possibly Alonzo Church. But it's still available in code today, which is really cool, and speaks to the continuing power or at least recognition of importance of functional programming.

As I said before, the idea with lambda is that you could write an anonymous function. Over here, in order to write a function that's squared, I had to write out a define line. And because Python uses indentation, because in Python, indentation carries meaning, I'd have to enter and return over to a next line-- I'm actually not sure if that's strictly true.

I think you can do def square x, return x, but I'm not positive. I've only seen it written like this. And I think it's because in the general sense, people try to respect things like convention and readability in Python.

If you're using lambda, people already know that you want to describe a function really quickly. Or you want to describe function without assigning it any name. Therefore, the two most common uses you'll see of lambda is when you want a really fast function and you don't want to spend extra time or lines writing out that function.

It's pretty clear what this is going to do. Or if, in a particular sense, you need an anonymous function. You don't want to assign a name or memory space associated with that function.

Note instead of using square, I can just write out this. So I saved two lines of code defining square. And then I don't have to refer back to some earlier part of the code in order to understand what this line does. Pretty cool.

All right. So there's functions of first class objects. There's lambda. Let's talk about how to use lambdas on lists.

In Python, you'll end up doing a lot of list manipulation. One of the best uses of anonymized functions in any functional programming language is the ability to manipulate lists in a line. If I've defined just a pretty straightforward list right here, I can use functions like map filter and reduce to-- in one line-- take a list, apply a function to every element in that list, and then return a copy of the result of that list.

I didn't have to write a separate function to do the list handling. I didn't have to write a separate function to multiply the list by two. And I've communicated what I'm doing

effectively to people that are used to functional programming.

So if you type this line in, you should end up with a return in interactive mode of every element in this list multiplied by two. Now's a good time to try. Once you've done that, I also recommend looking at the original copied demolist.

Note that this is unaltered. Map actually returned a new data structure that represents performing this function on this list. This becomes important later when I explain aliasing, but I will do that in about two minutes.

If we want to get even more elegant, we can use list comprehensions. List comprehensions-- if you ask somebody where list comprehensions are from, they'll probably say Haskell, because that is the most popular use of list comprehensions, in terms of pure lazy functional programming. But it actually goes back further than that, and is in Small Talk, and then something from the '60s. I can't remember the name of it right now.

They're really nice because they borrow a syntactic approach from mathematicians. This looks a lot like set notation. And when you read this, you can probably tell that the list listComprehension is going to describe a set of points from 1 to 4. But you accomplish that in possibly one line of code. I've written it on three here because I wanted to keep it in this space. But it's really concise.

If you type this into IDLE, you should see the kind of list that it returns, and also what's listComprehension is now-- what now constitutes listComprehension. You can use listComprehensions with functions like map and filter and reduce. And, along with the anonymized functions, all of these tools provide you with a lot of functionality in a very short amount of space.

It's also good to be able to recognize what's going on when you see something like this or something like this. Because you'll probably run into it in, in particular, a lot with Lisp code, but also in the artificial intelligence community in particular.

Ok. We've talked about all that. Let's take a second to talk about the fact that lists

are actually mutable, and what that means, and what things you have to be careful about if you're going to be working with mutable objects.

If you're new to programming, or you're new to Python, you've probably already worked with some of these data types, right? Any numbers, any strings, any tuples, are going to be immutable. What that means is if you have a variable and you have a second variable that has a different assignment line associated with it-- right? I haven't assigned g to h here. I've just signed g to "hello" and h to "hello."

If you look at the space in memory that Python associates with both objects, they point to the same place. This is the definition of an immutable object. When g points to "hello" and h points to "hello," they both point to the same place. If x points to 5 and y points to 5, they're both pointing to the same place. If you point x at 6, it now points to a different memory address.

This gets confounded when you're talking about mutable objects. The problem with mutable objects is that you select a memory space to contain the object or memory ID in Python to contain the object. And then that object is changed in place.

You can use this to your advantage but it can also mess with you. And here's how. Let's say I assigned a to a small list. And I also assigned b to a. b and a now point at the same place.

If I manipulate b, and I'm-- excuse me-- still working with an immutable object, or excuse me, if I'm still working with a mutable object, the object in that memory address has been altered. And b and a still point to the same place in memory. So if I look at a, it's going to look like b, which is going to look like 1, 2, 3, 4 -- which is not what I assigned a to originally.

Again, can be powerful, but you have to keep it in mind. And it might start to mess with, in particular, the 6.01 software when you're dealing with state machines.

In order to get around this, you can create a copy of the list and then modify the copy. This is actually what map does. If you want to do it, the easiest way to do it on the first layer is to specify the index into the list with no bounce. It'll copy the whole

thing.

There's also a Python library copy or deep copy, if you want to copy lists of lists of lists of lists of lists. And just to clarify, you'll note that after you assign c to a copy of a, c and a occupy different memory places. So if you modify c, a will retain its original value.

I think that's all the notable things I have to say about Python. and gives us enough power to do some really powerful list manipulation or array manipulation, and covers what we want to say about functional programming before we get into the notion of state, which I'll talk about next time.