PROFESSOR: Happy Valentine's Day 11. Actually, maybe it's a little smiley face combined with an 11. Did any of you leave this here for me? Or am I just stroking my ego and this was left for someone yesterday? Stroking my ego, all right.

OK, last lecture, we looked at a program for finding roots and put in a little debugging statement that, along the way, printed various approximations to the root. Now, suppose that instead of printing things, we actually wanted to collect the approximations. For example, to be able to go back and look at them later and analyze them, do various kinds of things.

To do this, and this is the sort of thing we do a lot, we need some data structures that can be used for amassing collections of items. There are 3 data structures in Python that are used to collect items. I'm going to try and cover all of them today. Tuples, lists, and dictionaries.

We'll start with tuples and lists. And what they have in common is they are ordered sequences of objects. So the key notion here is they're ordered. It makes sense to talk about the first object, the second object, the last object, et cetera. When we get to dictionaries, or dicts as they're spelled in Python, we'll see that they're not ordered.

All right, let's look at tuples first. They're the simplest. So if we look at it, there's a very simple example at the top. I have this tuple called test. And I just said that's the sequence of ints 1, 2, 3, 4, 5. I can then index into it. For example, look at the first element, which is the 0th element. Or I could look at the next element. And we can print them.

So let's just do that. And you can see it prints 1 and 2, not surprisingly. I can print

the whole thing if I want. That lets me look at the entire tuple.

I can also look at this. And that gives me the last element without my having to know what the last element is. I can ask about the length of a tuple. And it tells me it's five. Similarly, I could write something like this. Print test.

Why was that out of range? Yeah?

AUDIENCE: Because you're indexing from 0?

PROFESSOR: Because I'm indexing from 0. So that's why I have this more convenient way of writing minus 1. Otherwise, I'd have to do len minus one. Good grab.

OK, let's look at a little example of how we might use this sort of thing. So here, I've just written a little piece of code that finds divisors. Going to find all of the divisor of 100, collect them into a tuple.

Notice this kind of funny piece of syntax here, i comma. I need to do that to say it's a tuple of length one. Why can't I just write open paren, i, comma, open paren? Because that would just take the expression, i, and parenthesize it as we often use parentheses for grouping when we write things.

So by inserting this comma, I say, I don't just mean the-- in this case-- say the number i. I mean the tuple of length one. So it's sort of a special case piece of syntax that you need for tuples of length one. Then, I can print the divisors. So let's run that. And it now prints the tuple for me. So I've run through. I've computed all the divisors. And I've collected them. Nothing very interesting, but kind of useful.

We can also-- I've shown you how to select elements of tuples. I can also, if I choose, get what are called slices of tuples. So a slice gives me a range of values, or in this case, a subsequence of the tuple. As we'll see, we can also slice lists. So let's see. We have a tuple called divisors here. Yeah, OK, I'll save the source. Oh, come on!

So if I wanted to do-- what did I call it? I called it divisors. So I can do something like divisors 1:3. And you'll note that gives me those two elements at the appropriate

places. And so it's a very convenient way to take pieces of it. All right, any questions about tuples? Not very deep.

Lists are, I think, more useful than tuples and also, alas, more complicated. And they're complicated because the big difference is that tuples are immutable. And by that, I mean once you've created a tuple, you cannot change its value. You can create a new tuple. but you can't change the value of the old tuple.

In contrast, lists are mutable. Once you've created a list, you can actually change it. It's the first mutable type we've looked at because you'll recall assignment didn't actually change the value of an object. It just changed the object to which an identifier was bound.

Mutability is the first time we've seen a way to actually change the value of an object itself. And that's, as we'll see, both a powerful concept and an opportunity to mess yourself up by committing serious programming blunders. All right, so let's look at an example here, first of many we'll be looking at.

So first, we won't worry too much about the mutability. So here, I'm creating a list called techs, which happens to be, in this case, a list of strings. Lists need not be homogeneous. As we'll see, you can mix strings, and floats, and ints. And most interestingly, you can have lists of list.

And then, another list called ivies. And then, I'm going to say I'm going to have these univs, yet another list. This list empty, containing no elements. Then, I'm going to append techs to ivies. Notice this syntax-- univs.append.

What that means here is that append is what Python calls a method. As we'll see when we get to classes, methods play a very important role in Python. But to a first approximation, it's quite safe to think of a method as an alternative syntax for writing function. So instead of writing something like append a list and an element, I write l.append the element. And just think of this l over here as a fancy way of denoting the first argument to the function append, the first actual parameter.

When we get to classes in a few weeks, we'll see why it's highly useful to have this specialized syntax. But for now, just think of it as a piece of syntactic sugar, if you will. The thing I want you to think about, though, is this is not equivalent to assigning something to l. This actually mutates the list. And we say it has a side effect.

Up till now, since we've only been dealing with immutable types, every function we've looked at, its job was to take in a bunch of values, do some computation, return a value. It didn't change anything.

And then, if we wanted to take advantage of what the function did, we had to assign the value it returned to some variable, and then we could manipulate it. Or we could print the value it returned. We had to do something with the value it returned.

Here, we invoke append. And rather than worrying about what it returns, we're invoking it for the purpose of its side effect-- the modification it performs on the list. So let's look at what we get when we run this.

So you'll notice what univs is now. It's a list of length one. And the one element in it is itself a list because I have appended a list to the end of the empty list-- not the elements of the list, but the list itself, OK? So it's important to notice the difference between a list that contained the elements MIT and Cal Tech, and a list that contained a list which contains the elements MIT and Cal Tech. Yes?

AUDIENCE: Previously, you added two tuples together. And that's not like appending, right?

PROFESSOR: Because when I concatenated two tuples, in order to do something useful with that value, I had to assign it to something. It did not modify anything. It produced a new tuple which was the value of appending them. And then, it assigned it to a new tuple. So it's quite different from append, which is actually having a side effect on the list. Does not produce a new list, it modifies the old list.

So we can look at this, draw a little picture here. So we had techs. And that pointed to a list with two elements in it, which I'll abbreviated as MIT and Cal Tech. Both of these elements were strings.

Then, I created a new list called univs. And that was initially bound to the empty list, a list with no elements in it. I then did an append. And the effect of the append was to modify univs so that it pointed-- now, I had one element. And the element it was was this list.

Notice it didn't copy this list. It actually included the list itself. So let's look at the ramifications of doing it that way. Whoops.

So what I'm going to do now is append another element called ivies. I'm going to then print it. And then, for e in univs-- so here's kind of a nice thing you can do with lists. You can iterate over the elements in the list.

So you might think that the way to do that is, well, I'll go for i index in range 0 to length of list. That would be equivalent. But it's, in fact, much easier to just write this way-- for e in univs. That will do something to every element of the list. I'm going to print what the element is. So let's look at that.

So now, I have a list, as we see here, of length two. It contains two lists. And if I print the elements, I print each of those lists. Nothing very magical there.

Now, suppose I wanted it flattened. You asked the question about the tuples where I did concatenation. Well, I can do the same thing here. I'll let flat equal techs plus ivies. And then, I'll print flat. And you'll note here what concatenation does is it just takes the elements of the list and creates a new list and appends it. Not append it-- in this case, it assigns it to flat, excuse me. So that's convenient. Poor old plus is overloaded with yet another meaning. All right, let's keep on trucking.

I, of course, can do this myself. So here, I've got another list called artSchools. It includes RISD and Harvard. For u2 in artSchools, if u2 in flat, I'm going to remove it. All right, so again, I'm going to iterate over everything in artSchools. What's this going to do, do you think? What will I get when I print it here?

No new concepts here. This is all stuff we've seen. Somebody up there? Yeah.

AUDIENCE: Flat without Harvard in it?

PROFESSOR: Yes. The correct answer was flat without Harvard in it. Wow, almost a good catch. Just almost there. All right, so let's confirm. Yes, we'll save it. Thank you.

All right, so we've now removed the art school. All right, we'll look at one more thing. Actually, we'll look at far more than one more thing. But we'll look at one more thing for the moment.

So I'm going to invoke another method. This is a built-in method of Python that works on sequence types. And it's called sort. So you can do-- this will have a side effect on flat. And it will, as you might guess, put them in order. So let's run that. Actually, we'll comment this out for the moment.

So you'll now note that it's put them in alphabetical order. This is something, again, that you'll find convenient throughout the term, the ability to have the side effect of sorting a list. Now, I'm going to assign something to flat sub 1. So let's think about what this is going to be doing.

It's going to replace the first element of flat by a new value. So it's having, again, a side effect on flat. So you have to be a little bit careful as you think about this, that I've written something that looks like a conventional assignment statement. But in fact, flat sub 1 is not an identifier. So this is not binding the name, flat sub 1, To UMass. It's actually modifying the object that is the first element of flat.

AUDIENCE: So would an identifier just be a name, flat, for example?

PROFESSOR: Well, I think the way-- the question is, would the identifier just be "flat?" No. Don't think of this as an assignment at all. Because remember, what an assignment does is swing one of these pointers to point to a different object. We'll see an example of that. Whereas here, I'm actually modifying a piece of the object that the identifier points to. So it's not an assignment in the sense of a re-binding. It's actually having a side effect of modifying the object.

Let's just run it. And then, I'll be happy with the question. So here, you'll see I changed flat sub 1 to now be UMass. Yeah, question?

AUDIENCE: As you've drawn on the blackboard here, you've drawn an arrow to the actual object that's techs. So if we change techs, will we change univs?

PROFESSOR: Yes. The question was if we change techs, will we change univs? Now, in some sense, that's a philosophical question. From the philosophical point of view, maybe, you could say no. Univs is still the same object. So the binding has not been changed. But the object to which it has been bound is now different. Same object, but it has a new value, all right?

So this is the key thing to keep in mind. Assignment has to do with the binding of names to objects. Mutation has to do with changing the value of objects. We'll see that pretty graphically in the next example that I wanted to work my way through.

So I'm going to work through a dull example. But I think it illustrates the points. And I do, by the way, very much appreciate the questions, even if I forget to throw you candy in return for asking. But it's good. If you have questions, please do ask them. And I'll try and remember to feed you.

So let's work through what this code is going to do. So first, I'm going to have the list L1. So the first thing, I'm going to create an object which is a list of length one containing the integer 2. So L1 will be bound to the list of length one containing the int 2.

I'm then going to create another list, L2, which is going to be of length two. And the first element will be L1. And the second element will be L1. So what will happen if I print L2? What will I get?

I'll get list two, comma, list two, right? We can look at it. If I print L2, excuse me. All right, just what we would have expected.

Now, I'm going to change the 0th value of L1 to be 3. So I'm going to mutate L1. Now, if I print L2, I will get a different value, 3, 3. Not surprising.

It's something to keep in mind that can be useful, but can also be confusing. Because if I'm looking at my code, it doesn't look like I changed L2 when I have a

side effect on L1. And so it can be mystifying when you're trying to debug. You print L2. You do a bunch of-- execute a bunch of statements, none of which apparently deals with L2. Then, you print L2 again and get a different value. This is both the beauty and the peril of mutation.

Now, let's see what this does. So here, I've now mutated L2 so that its first element is no longer the list, L1, but is now the string a. And we'll just do a bunch of these all at once here.

Now, I'm going to change L1 to be 2, length one. So what do you think will happen, by the way, if, after this, I print L2? Whoops. What do you think is going to get printed here? This is important. You need to figure this out. What's going to get printed here? A volunteer, please. Yes?

AUDIENCE: it's going to print "a" in the first slot and then L1 in the second slot.

PROFESSOR: Well--

AUDIENCE: So it's "a" and then "2."

PROFESSOR: "a" and then "2" is one conjecture. Let's find out. "a" and then "3." Why?

Because what happened here is, when I did the assignment to L1, what that effectively did was swing this pointer to point to the new list containing the element 2, but had no effect on this object. I was changing the binding of the identifier. I was not mutating this object. And so this element still points to the same list, which was not mutated. That makes sense now? So you have to get your head around the way all this stuff works. Anybody have a question about why this is what it did? All right, if not, we'll just roar right along.

So now, we can do various things. And we'll get some stuff. All right, moving right along.

Here's an interesting little program not in your hand out. Let me get rid of all this cruft. So here's a function, copylist. It takes a source list and a destination list. And for e in the source list, it appends it to whatever the destination list used to be. And

then, I'm just putting in a little print statement so we'll be able to see as it runs what it's doing.

So I'm going to say L1 is equal to the empty list. L2 is 1, 2, 3. And I'm going to copylist L2 to L1. Print L1 and L2. So what am I going to get when I print those things?

This is the easy question. Don't tell me I have you so intimidated that you think I'm playing. This is not a trick question. Pardon?

AUDIENCE: 1, 2, 3 for both of them?

PROFESSOR: 1, 2, 3 for both of them. Who said that? Raise your hand. Someone back-- oh, good grief, all the way in the back. All right. Oh no, not even close! Off by a row, and about four people, too.

OK. so let's try it. 1, 2, 3, 1, 2, 3. So exactly as predicted.

Now comes the trick question. What is this going to do? Pardon? Well, we'll print L1 here. What do you think it'll do when it gets to that print statement? Will it get to that print statement? Let me ask that question. Will it ever get there?

AUDIENCE: No.

PROFESSOR: No. Bingo. Let's run it. And you'll see LDest just gets longer and longer and longer. Why is that happening? Because what it's attempting to do is look at LSource and copy the remaining elements of LSource to LDest. But every time I go through the loop, what we have is a situation where the formal LSource and LDest are now pointing to the same object.

So every time I modify LDest, I am modifying the object to which LSource points to. And I keep finding yet another thing to copy. This is an example of what's called an alias, one object with two names, or in general multiple names.

When you have immutable objects, aliasing is perfectly harmless. If you have 58 different names for the object 3, it doesn't matter because you can never change

what 3 means. Here, where you have multiple names for the same mutable object, you can get massive confusion when you modify it through one name and then forget that it's being accessed through another. So it's something to worry about.

As with tuples, you can slice lists. You can index into lists. You can concatenate lists. You can do all the usual things. I'm not going to list all of the operators. And there are a lot of very nice operators. But we'll post readings where you can find what operators are available.

I now want to move on to the third built-in type that collects values, and that's a dictionary. A dictionary differs from a list in two ways. One, the elements are not ordered. And two, more profoundly, the indices need not be integers. And they're not called indices. They're called keys. They can be any immutable type. So we'll look at a simple example first.

So here, I'm creating a new dict. We use set braces rather than square braces to remind ourselves that the elements are not ordered. And I'm saying the first key is the number 1. And that's bound to the string object "1, 1." And then the second key is the string object deux, which is bound to the string object 2. And the third one is the string object pi which is bound to the float 3.14159.

Now I can then index into it. So for example, if I choose to, I can write something like print d sub pi. I'll just stop it here with my old trick of asserting false. And you'll see it will print the value with which the key is bound-- to which the key is bound.

So what we have here is a dict is a set of key value pairs. And I can access it by looking at the keys. I can do an assignment, d1 equals d, just as I can with lists. And now, I remember this is a real assignment. So now, I have an alias, two identifiers pointing to the same dict. I can then print d1 sub 1, either print it or I could assign to it. So let's do that. First, we'll print it just to show what we get.

Then, I'm going to do an assignment, saying, OK, I want to now change the binding in d of the key one to be uno rather than one. And we'll get something different. Let's just run this. So you'll note, as we would have guessed, with mutability, we see

it showing up. So far, just like lists, the difference being we have key value pairs rather than you could think of a list as being int value pairs. The indices of a list are always ints. Associated with each of those ints, we have a value. Let's look at a more fun example showing what we can do with dictionaries.

So here, I have a very simple dictionary called EtoF, for English to French. And we can do some things with it. So I can print it. And that's kind of interesting. Let's see what we get when we print it.

So it's printing it. And you'll notice the order in which it's printed, the key value pairs, is not the order in which I typed them. That's OK. The order in which it prints them is not defined by Python. So there's no way to predict the order. And that makes sense because, by specification, dictionaries are unordered. They're sets, not sequences.

I can print keys, EtoF keys, open, close. Keys is a method on dicts that returns the keys. And then, just for fun, I'm going to try to print EtoF.keys without the open, close. So you'll see when I first printed it with the open, closed, I got all of the keys in an order that was not necessarily predictable. Certainly not the order in which I typed them.

But that's nice. I now have this sequence of keys that I could do things with. When I typed it without the open, close, and I wanted you to see this, it just says it's a method, a built-in method. It does not execute the method, right? That's fine.

But again, you have to be careful that you'll find this happening when you write code. You'll forget to write the open, close. And you'll wonder why your program isn't doing what you expect it to do. Remember, it's not saying that the method, keys, has no argument. It has one argument. In this case, EtoF. And the open, close is basically saying, all right, call the method rather than the method itself. Again, this will be very important when we get to classes. Let's get rid of some of these print statements now.

Suppose I want to delete something. I can delete something, so delete EtoF sub 1

del is a command, much like print, that says remove something from EtoF. So I can do that. And now, something has disappeared. The key value pair where the key is one is gone. So that's how I remove things.

AUDIENCE: Question.

PROFESSOR: Yes?

AUDIENCE: Are the keys of dictionaries mutable?

PROFESSOR: The keys have to be immutable. You cannot use a mutable type for a key. We'll see the reason for that in a couple of lectures when I talk about how dictionaries are implemented in Python. They use a very clever technique called hashing which would not work if the keys were mutable. So in order to get an efficient implementation of dictionary look-up, we need to have immutable keys. And so that's required.

Let's look at another example. So here, I'm again setting a dictionary. And now, what I'm going to show you is that I can iterate through the keys and print the values. For key in d1.keys, nothing very fancy here. OK? one equals uno, pi equals 3.1459, and deux equals 2. All right? So nothing very dramatic happening here.

Finally, just to illustrate why this sort of thing is particularly useful. And in fact, you'll find it quite useful in the upcoming problem set. I'm going to do some translations. This, by the way, is not the way Google Translate works. It's a bit more sophisticated.

So I'm going to have two functions, translateword, which, if it finds the word in the dictionary, returns the value associated with the key. So note, if word in dictionary says is there a key whose value is word? If so, return the value associated with that key. Otherwise, don't even try and translate it. Just leave it the way it was.

And then, translatesentence will set the translation equal to the empty string and wordequals to the empty string. Then, it'll just collect all the characters until it finds a blank, translate the word using translateword, and append it. And when it's done, it

will print it. So we can now do these translations quite simply. What could be easier? You never again have to learn a foreign language. Just use Python to do the translations for you.

All right, that's it for today. Remember, there is a problem set you should be working on and another one that will be posted. Take care.