PROFESSOR: Good morning, everybody. Let's go back to where we were. We were talking about abstract data types and the notion of object oriented programming. And in particular, the role of classes in all of that.

A lot of people think this notion of object oriented programming is a new notion. In fact, it's not. It's been around for probably at least 35 years, maybe even longer. But it's only been widely practiced for maybe 15 years. Even that to most of you will seem like a long time.

It started in the mid 1970s when people began to write articles explaining this advantage of the approach to programming. And at about the same time, the language Smalltalk was developed at Xerox Park and CLU developed at MIT.

And they were the first languages that, in an elegant way, provided linguistic support for this style of programming. But it didn't really take off in the public until the introduction of Java, considerably later. And Java was the first popular language to support object oriented programming.

After that there was C++, which supports it in not a very elegant but a usable way. And today probably Python is the fastest growing language supporting object oriented programming. It's used widely, and that's one of the reasons we teach it here.

As I said, at the bottom of it all, the most fundamental notion is that of an abstract data type. And the idea is that we can extend our programming language by adding user defined types. And we'll shortly see why that's such a useful thing. And that these types can be used just as easily as any of the built in types.

Why do we call them abstract data types rather than just data types? We do that because we are essentially going to define for each type an interface. And essentially, what the interface does is it explains what the methods do.

What do I mean by what they do? What they do at the level of the user, not how they do it. That, of course, is the way the built in types work.

It wasn't until Tuesday that you understood how Python made dicts do what they do. Before then, I just explained, that you could put in a key and a value, and you could look it up and do an associative retrieval. It was maybe not magic, but it was wonderful that you could do this and not have to bother your heads with how it was made to work.

And that was because we provided, or the designers of Python provided, an interface that lets you use it. We'll do the same thing with the abstract data types. The key idea here is one we've talked about before and that's a specification.

It is the specification of a type, or of a function, or of a method, that tells us what that thing does. And we'll now until the end of the term try and maintain a very clear distinction between specifications and implementations.

Let's look at an example. So the example should be familiar to you. You will remember that on Tuesday at the start of the lecture, we looked at how we could use hashing to implement a set of integers. And I explained to you that I wasn't very happy with the implementation because of the way it used this global variable.

Now we're going to see a much more elegant approach. So I'm going to define a new abstract type called intSet. I do that by writing the word, class, followed by the name of the class. And then there's this funny thing saying that it is a subclass of objects. Ignore that for now. And I'm going to come back to it later.

But fundamentally what it's saying is that every instance of intSet is an object. That's not very interesting because everything in Python is an object. So from an information theoretic point of view, there's no information here. Later we'll see that we can use this mechanism in a more interesting way.

Now let's look at the methods. So first I tell you a comment, it's a set of integers. Then I've got this funny method called underbar underbar init. Whenever you see something with an underbar underbar in its name, it has a special status in Python that lets us do elegant things with the syntax.

What will happen every time I create a new object of type intSet, the underbar underbar init method, or function, of the class will be executed on that object. What it will do, in this case, is introduce two attributes of the object.

The attributes are numBuckets which is now replacing the global variable we looked at last time. And I've arbitrarily chosen 47. And Vals. This will be the hash table itself containing the values. And then, exactly as we did on Tuesday, I'm going to initialize the values so that each element of this list is now an empty list.

Now what's going on with this funny notion of self. Let's look at an example here. I'm going to say s equals intSet open close. This will create a new intSet object, execute the underbar underbar init.

So if, for example, I print self.numBuckets-- whoops, can't do that because self is not defined in this environment. Self was a local variable to underbar underbar init. In fact, the formal parameter.

But I can write s.numBuckets, and I'll see it's 47. NumBuckets and Val are now attributes of s. Attributes of the instance s of the class intSet.

And I [[UNINTELLIGIBLE] what we would expect. Yes?

AUDIENCE: You didn't put any object in between the parentheses.

PROFESSOR: Yes. So that-- the question is well, it looks like underbar underbar intSet or underbar underbar init has a formal parameter. And I've given it no corresponding actual parameter.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes. Let me finish. So that's what it looks like. And that's the magic of this syntax. It automatically passes an implicit object, or it creates one. It's a very special role for underbar underbar init.

Self is used to refer to the object being created. I'm going to come back in a minute to discuss more fully the concept of self and how it's used here. So just give me a minute to get there.

The next thing we see is hashE. This is something we looked at again on Tuesday. It's a private function in this case that I don't intend to be used outside the class.

So if we think of the specifications here, the interface of the class, it does not include hashE. That's what private means here. This is a convention. It's not enforced by the language.

And unfortunately, as we'll see, a lot of useful things are not enforced by the language. But nevertheless, good programmers follow the conventions. And we expect you guys to do so as well because, of course, you're good programmers. So this doesn't do anything very exciting. It's just what we saw.

Insert is more interesting. Let's look at what that does. It apparently takes two arguments, the formals named self and E and inserts E into self.vals. However, if we go look at the code that uses it in say testone what you'll note is I'm saying for i in range 40-- this is just like the testone we looked at Tuesday. I'm going to say s.insert of i.

It looks like I'm calling insert with only one argument, but as we discussed last time, this s before the dot is actually the first argument to the method insert. And so it's getting two arguments really. And by convention, that implicit first argument is always called self in Python.

It's not enforced. You could call it George if you preferred, or Alice, or whatever you like. But if you do, you will confuse the heck out of anybody who ever reads your code, including any TA you ask for help. So please do use self.

It's just a name. It has nothing to do with what a philosopher, or a psychologist, or an ethicist might think as the concept of self, this wonderful elevated concept. It's none of those. It's just a name. But stick to using that name.

So if we go back, I can insert a bunch of things. I'm going to then print s. That's kind of interesting. UnderBar underbar STR, underbar underbar is one of those special names as well.

The code is pretty simple. All it is is returning a string representation of the set. And it could be anything you want. I chose sort of a conventional way of denoting sets.

What's interesting is that it gets automatically called by print. So when I write the command, print s, in test one, the Python interpreter is smart enough to know oh, I better take s, convert it to a string, and then print it.

How does it know to convert it to a string? It automatically invokes the underbar underbar STR method.

And then the other operation is member, which again is exactly what we looked at before. But you'll notice in this code that uses intSets, I make no reference to the data attributes of the class directly, or I shouldn't. You'll notice that I wrote evil next to s.vals.

Python will let me do it, but I shouldn't. Why shouldn't I do it? Why am I saying this is evil? Well, would you be happy if you got a message saying IDLE was changed. Please download a new version and the new version happened to have a different implementation of lists or dicts. And it caused all of your programs to stop working. You would be pretty unhappy.

Now why won't that happen? Because your programs don't depend, in any way, on the way in which people chose to implement those built in types because you programmed to the specification of the types, not to the implementation.

The specification of intSet did not mention Vals or numBuckets. Therefore, as the implementer of that class, I'm entitled to go back and change it. So I'm not going to

use a hash table at all. I'm going to do something else. I'm going to use a red, black tree, or some other fancy implementation.

I'm allowed to do that. And if I make that change, and Vals disappears and numBuckets disappears, your program should continue to work so long as I still meet the specification.

The minute you go in and directly access the variables of the class, those attributes, you have used things that do not appear in this specification. And if I change the implementation, your program might break. So you shouldn't do that. Does that make sense to everybody?

It's a very important concept. The concept is known as data hiding. It is really the most important development that makes abstract data types useful. It gives them the same status as the built in types.

The minute you choose to ignore this, you do so at your own peril. Some programming languages, like Java, provide a mechanism to enforce data hiding. The designers of Python, for reasons I do not understand, chose not to. I think it is a flaw in the language.

The things we're hiding are the instance variable. Those are the variables associated with each instance of the class.

We should also hide class variables. We haven't seen those yet, and we'll see them later. The instance variables, we get a new copy of each time we created a new instance of the class, a new intSet in this case.

The class variables are associated with the class itself. And you get only one copy of them. Later on, we'll see an example where class variables are useful.

All right. So let's run this just to make sure it works. It does what we would expect. True/False. And then you'll see this nice string representation of the set. And then just to show you what happens when you do the evil thing, I printed as we did last time the actual list that came out.

Let's now look at a more interesting example. And the idea I want to convey here is how we use classes and abstract data types to design programs. So imagine that you're writing a program to keep track of all the students, faculty, and maybe staff at MIT. It's certainly possible to write that program without using any classes.

For each student, you might give them a family name, a given name, a home address, years, grades, et cetera. And you could do this with some complicated combination of lists and dictionaries. But it wouldn't be very elegant.

So what I want to do before writing that program-- and I won't actually write that program. I'll just write some of the classes we can use-- I want to pull back and think about what abstractions would be useful. So this style of programming in which you organize your programs around abstract data types says before we write the code in detail, we think about these types that would make it easy to write the code.

So for example, if you were a finance student, and you wanted to write some code dealing with markets, you might want to have an abstraction of a government bond, and another abstraction of an equity, and an extraction of a call option. Whatever you want it.

But you'd say I want to think at that level of abstraction. I don't want to think about lists, and dicts, and floats. I just want to think about options, which have a strike price, a date, and things like that.

Similarly, as I'm working on this database for MIT, I want to think about abstractions of students, and faculty, and staff. I'm also going to use what's called inheritance to set up a hierarchy of these.

And the reason I'm going to do that is I want to be able to share a code. I know that there will be certain similarities between students and faculty. Some differences too.

But I want to begin by saying what's not different? What's similar? What's the same? So that I only have to implement it once and get to reuse it. So if I pull back and say is there an abstraction that covers the shared attributes of students, and faculty,

and staff, I might say it's a person. They're all people. And it's arguable whether every faculty member is a human being, but for now let's pretend.

And so I'm going to start with this abstraction person. I'm going to import something called DateTime. I'll show you what we're doing when we get there. But it's like we've imported math before. This is a class somebody else wrote that deals with dates and time in a fairly reasonable way.

So I'm going to import that here into person. Probably didn't need to import it twice. Maybe I'll just simplify it by getting rid of this one.

UnderBar underbar init here will create a person with name 'Name'. And you'll notice what I'm doing here is introducing an extra attribute lastName -- self.lastName. And that's because I just want to make life easy for myself. I figure I'll want to find the last name frequently. Let's once and for all get it and put it in something.

And I'm going to initialize birthday to none. The next attribute, or the next method, is get lastName. I have that here because I don't want users of this abstraction to even know that I have an attribute self.lastname. That's part of the implementation. And so I have this method that fetches it.

And you'll see as you build classes that you often have things called get. And those are typically methods that return some information about an instance of the class. You'll also frequently have set methods, for example, set birthday that gives values to instances of the class.

More interestingly, I have get age. And that's using some of the built in operations on DateTime conveniently. It allows me to subtract one date from another and get a number of days. So this will allow me to return somebody's age in days.

Then I've got another underbar underbar method we haven't seen yet. It stands for less than. Not a big surprise. And I'm going to use this to order names, or order people. Why am I using a special operator rather than just putting in the method L-E-S-S-T-H. A-N? E-N?

I'm doing that because I want to be able to write things like p1 less than p2 in my code. And Python will take that and turn it into the underbar underbar LT. Better yet, if I have a list say of person's, I can use the built in sort operator on that list, and it will be smart enough to know when it's comparing two people to do the sort to use the underbar underbar LT. Very convenient kind of thing.

Let's look at an example. Let's look at some code. So I'm going to set me to John Guttag, and him to Barack Hussein Obama, her to Madonna. And we'll print some things.

So I printed him, and I printed him.getlastName. I can now set some birthdays. Let's look at this line. How do you feel about him.birthday equals 8/4/61?

We'll come back to it. But I want you to think about it. And we see that Obama is-- well, we see their ages here. Actually, we see that Madonna is older. She looks really old when you look at that number of days, doesn't it. Maybe she is.

Now what's going to happen here? I messed up. And I messed up because I went in and directly accessed the instance variable and assigned it what I thought was a reasonable representation of a birthdate. But I shouldn't have because that's not even the appropriate type. It's a string rather than something from DateTime.

So again, we see the impact of my having done this evil thing of violating the abstraction boundary and stuck in there try to directly access an instance variable. We can do some comparisons.

And what we see is that I'm not less than Madonna. I guess that's OK. You with me so far?

I can make a list of these things and print the list. So it does a fairly nice job calling the underbar underbar STR operator.

And you'll note I had no trouble throwing objects of type person into the list. No different than putting ints, or floats, or any of the built-ins. So it all works nicely.

And then I can sort it and print that. And now the lists come out in a different order because it's using the underbar underbar LT operator to sort the elements.

All of this is just by way of showing you how convenient it is to write code that uses a data abstraction. If we go back and look at the code, we'll see that once again person was a subclass of object. That's why we can do all these things we've been doing with it.

But now I'm going to start using the hierarchy. MIT people are special. I hate to say that because I know we have non-MIT people in the room. But MIT people are special. Well here's at least one of the special attributes of MIT people. They all have an ID.

So I'm now going to say an MIT person is a special subclass of person. So it has all of the properties of a person. And the way we describe that is it inherits the properties of the super class. And it adds a property.

We can now assign an ID number. And now we're going to see the thing I promised to show you, which is a class variable. Next ID num is not associated with an instance of MIT person, but it's associated with the class itself. It's a class variable.

I can do that because classes are themselves objects. And the advantage of this is every time I get a new instance of this class, I can assign a unique ID. Similar to the way we were using global variables earlier in the term, typically once we have classes, we'd stop using global variables because we have these class variables, which can serve very much the same purpose in many cases.

And so now every time I get a new MIT person, I give them an ID and then increment the ID number so that the next person will get a different one. So I added a property, this ID number property, which I find by get ID num.

I've also overwritten an existing property. You'll note that I've changed the definition of underbar underbar LT. So it's now saying we're going to compare two people not on the basis of their names but on the basis of their ID numbers.

So let's look at some code that uses this. Get rid of the other code so it doesn't clutter up the screen. So we'll look at some things here. We'll get some MIT people called p1, p2, and p3. And we'll print their ID nums by calling get ID num. And you can see that Barbara Beaver gets 0, and the first Sue Wong gets 1, and the second Sue Wong has ID 2.

I can create even a third such person. And now let's think about what happens here. I'm going to print whether or not p1 is less than p2 and whether or not p3 is less than p2. What should we get when we do this? Somebody? Pardon?

AUDIENCE: True and false.

PROFESSOR: I heard a true false. And indeed that's right because it's comparing IDs. Now, suppose I want to compare names. I could if I chose do this. I could call person underbar underbar LT.

And what this says is don't use the less than of the subclass. Go up and use the super class one to do the comparison. So it's going up and doing that.

I can do other things. I can compare things for equality. Let me just rip through all of these and see what we get. Whoops. Surprise! Well, before we get to that surprise-- and it's not actually a surprise. I shouldn't have uncommented it. Let's look at the other ones.

We can say is p1 equal to p4? And we discover it's not. That's good. And we can say is p4 less than p3? That all works. It's not. But I can't say p3 less than p4. And why can't I do that? Why did I get an error message when I did that? Yes? Someone wanted to answer that?

AUDIENCE: That's because you mix [UNINTELLIGIBLE]?

PROFESSOR: Can you say that more loudly?

AUDIENCE: She's a person, not an MIT person. So you didn't assign her an ID number.

PROFESSOR: Exactly right. The answer-- I hit somebody right on the head. Now there's going to

be a traumatic brain injury. I'm going to get sued. It's going to be messy. Time to leave the country.

All right. Because it looks at p3 less than p4, looks at the first argument, which is p3, and says OK, what's the underbar underbar LT associated with p3? It's the one associated with an MIT person. Let's go execute that. And then it tries to retrieve the ID number of p4, which is not an MIT person, and it gets an error message.

All right. Nothing subtle. It's the same kind of thing we've seen all along when we use type errors. In this case it's called an attribute error because we've attempted to access an attribute of an instance that doesn't exist. And so we could catch it. It's raised an exception. We could catch it as we looked at Tuesday, but we're not going to do that because it's really a programming bug when that happens.

OK, let's continue. We were interested in students. So we're going to continue our hierarchy here. And now I'm going to introduce a subclass of an MIT person called an UG, short for undergraduate.

Underbar underbar init is going to call MIT person underbar underbar init, which will give the UG an ID number and a name. And it's going to introduce yet another instance attribute, or field, called the year. And so the year, initially, is none.

Then I can set the year. Though if I try and set it as something greater than 5, I'm going to raise an overflow error called too many. No undergraduate should be a year greater than 5. And I can get the year.

Let's look at what happens when we do that. So I'll have two UG's. Both happen to be named Jane Doe. And then the same MIT person as before. Let's run this code and see-- well, what's going to happen? What's going to happen when I say print UG1?

The first thing it's going to do is going to say is there an underbar underbar STR associated with UGs? And the answer is no. That's OK because I know an UG is also an MIT person. If I don't find it at the lowest level class, I'll bounce up and say all right, is there an underbar underbar STR associated with an MIT person? No.

That's OK.

I'll go up another level and say, well, I know an MIT person happens to be a person. And then they'll say oh, good there is an underbar underbar STR associated with person. So I'll use that one.

So it looks at the class. If it doesn't find it, it goes to the super class. If it doesn't find it, it goes to the super class. And it does that all the way up until the end, where at worst, it will use the built-in thing for printing objects because remember everything is a subclass of objects.

You don't want to do that. You want to have something more elegant than object at location xe345, or whatever if would have printed.

Then we'll do some comparisons. It will first look for the most local and then work its way up as needed. OK?

Let's keep going. We're going to introduce another kind of person called a graduate student. And I'm going to write pass. What that means is a G is an MIT person with no special properties, all the usual properties of an MIT person. Does not have a year because graduate students could be here more or less forever, which goes like this when I say that.

Why did I introduce the type in the first place? Because it lets me do type checking. I can now check whether or not a person is a graduate student because an instance of G will have all of the properties of an MIT person, but it will have a different type. It will be type G.

And so I can now ask the question is the type of this object a G. Or is it an MIT person and get a different answer. So I can do this. And if I go type of G1-- well, that's interesting. It says classmain.G.

It's going to upset its class. And the class is defined at the outermost level, which is called main. And the classes name is G, which is what we expect. And so I could write something like that.

Type of G1 equals equals G. And I get True. Ok. So it's a handy thing to be able to do. And in fact, I'm now going to bounce back to MIT person and add another method to it.

And this happens all the time when I'm programming that I go define a class, think I'm done, and then some time later decide that it would be convenient to add something new, another method. In this case, I'm now adding the method is student, which returns type of self equal equal UG or type of self equal equals G.

So this will let me distinguish a student from another kind of MIT person. So for example, if I want to have a course list-- another class, this is a subclass of object. You'll note I have an add student method, which takes self and who, maybe it should be whom. And it says, if not who.isstudent raise type error not a student.

So I'm getting some leverage now out of this. Now I wouldn't need to necessarily do this. I could have said if not type of who equals G, or type of who equals UG. But I chose not to.

The reason I chose not to is looking ahead, I might want to add some other students. I might want to add special student, for example. Or I might want to add cross registering student as separate types.

Now, the nice thing is I don't have to make a lot of changes. I know there's exactly one place in my code that defines what it means to be an MIT student. I go back and I change that method. And even if I asked whether somebody's a student in a 100 different places, I only have to make one change to fix my code.

So I'm getting some modularity by associating the method with the class MIT person rather than every time I need to use it.