PROFESSOR: I want to finish up our little excursion into searching and sorting, a very important topic in computing. How do you sort lists or databases? How do you search them? And I want to finish it up with hashing.

Hashing is how dictionaries are implemented in Python. And this leads to a very efficient-- at least most of the time-- search. But it comes at the cost of space. So this is an example where one can trade space for time.

So I want to start with a simple example to explain hashing. And I'm going to begin by assuming that what we're hashing is a set of integers.

So we want to build a set of integers and detect whether or not a particular integer is in that set. And we want to do that quickly. So the basic idea is we're going to take an integer-- call it i-- and we're going to hash it. I'll tell you what that means in a minute.

And what a hash function does is it converts i to a different integer, perhaps, in some range. So it, say, converts i to some integer in the range 0 to k, for some constant k. We're going to then use this integer to index into a list of lists. So each of these is called a bucket. And a bucket will itself be a list. So this together is called a bucket.

We've already seen that we can find the i-th element of a list in constant time. So when we ask whether some integers in this set will hash it, we'll go immediately to the correct bucket, the bucket associated with that integer. And then we'll search the list at that bucket to see if the integer is there. If this list is short enough, it will be very efficient.

All, right I realize that's very abstract. But let's look at the code, which will make it much less abstract. So the code starts with something very ugly that I'll apologize for. But very soon, we'll see how to get rid of that. I'm using a global variable here to say how many buckets there are going to be. And I've very arbitrarily chosen 47.

I then have a function called create, which uses this global variable and creates a list of lists, each element of which is empty. Because initially, we have no elements in the set. When I want to insert an element in the set, I'll call the function insert, which will hash the element. It doesn't actually even need to use the global numBuckets in this case, in fact. And then append it to the correct list.

So it calls this function hashElem, which could hardly be simpler. It just takes the remainder, the modulus of the element and the number of buckets. So that will give me a value between 0 and numBuckets minus 1. It gives me one of the list, and I'll just insert it at the end.

When I want to check for membership, you'll see it's quite simple. All I do is ask the question, is i in the list associated with the correct bucket? Remove is a little bit more complicated, but in fact, we don't need to spend much time looking at it now. It's just some code. And the only reason it's complicated is Insert doesn't look whether or not the element is already there. So it may occur multiple times in the list. So I would have to remove each one of them.

People see the basic structure of this? Why a list of lists? Why don't I just, say, have a list of Booleans, where I hash the integer, and it's true or false, whether or not I've seen it, depending upon the value of that bucket? Why can't I do that? Somebody?

Well, what's the property of this hash function? The key issue here is the hash function -- is many-to-one. That is to say, an infinite number of different integers will hash to the same value. Because after all, I have a set in which I can store any integer-- or any positive integer, at least-- and there are only 47 buckets. So it's pretty obvious that many integers will hash to the same bucket.

When two different elements hash to the same bucket, we have what is called a collision. There are lots of different ways to handle collisions. What I've shown you here is probably the simplest way, which is called linear rehashing. I'm not actually rehashing. I'm just keeping a list.

Does that makes sense? Yes, thank you for-- I'm glad somebody has a question. You have to ask loudly.

AUDIENCE: When you take the modulus 47, what does that return?

PROFESSOR: 0.

AUDIENCE: So hashElem always returns 0?

PROFESSOR: Well, no, sorry. It depends what I'm hashing. Sorry, I thought if you were saying that if I asked 47 mod 47. If I take 48 mod 47, I get 1. If I take 49 mod 47-- it's the remainder. Maybe I should have called it just the remainder.

Look up. I'm about to throw something at you. Ooh, I threw a curve ball.

OK. What's the complexity here of the membership test? Kind of hard to analyze. Roughly speaking, or exactly, it will be the length of the bucket, the size of the bucket. Now, I don't know how many elements will be in the bucket. But what will this depend on?

It will depend upon the number of buckets. If I have a million buckets, I'll get a lot fewer collisions than if I have two buckets.

So let's look at an example here. There's a small program called test. I said numBuckets to 47 in this case. And then I'm going to create it, create a set. And then I'm going to put a bunch of integers in it, then a few more, just for fun, including one very big integer, just to show that it works.

Then I'm going to show you what the set looks like. And in fact, what we'll do is we'll stop it here and see what we get. So what we'll see here is, as you would expect with that number of buckets, each of the small numbers hashes to a separate thing. That's just the way remainder works.

Not surprisingly-- in fact, it would be disappointing if 325 didn't have the same value both times I inserted it. So we say we happen to have one bucket that's got two elements in it. Happen to be the same. But this very big number happened to hash to the same value of 30 as 34. So here we have two elements in it.

A good hash function has the property that it will widely disperse the values you hash. So they end up in different buckets, rather than some stupid hash function that tends to put everything in the same bucket.

Now, let's see what happens if I change the number of buckets to, say, 3. Well, not surprisingly, we get some very big buckets, because there are relatively few choices.

So what we see here is we have a genuine trade off between time and space. If the number of buckets is large relative to the number of elements that we insert in the table, then looking at whether or not an element in it is roughly order one. Because these lists will be very short.

So we can actually look up something in constant time if we dedicate enough space to the hash table. If the hash table is very small-- the reduction ad absurdium case of one bucket-- then it's order n. It's not constant time. It's linear in the number of elements.

Typically, when people use hash tables, they make the hash tables big enough that for all intents and purposes, you can assume that looking something up is constant time, order 1. And that, in fact, is what Python does with dictionaries. It hashes the keys and chooses a big enough table so that looking up whether or not something is in a dictionary can be done in constant time.

If it then notices that the table is too small, because you've ended up putting a lot of elements in it, it just re-does it and gets a bigger table. So hashing is an extremely powerful technique. And it's used all the time for quite complicated things.

Now is it useful here only when we want to store ints? No. It would be kind of bad if that were the case.

In fact, any kind of immutable object can be hashed. Now, you may have wondered why the keys in dicts have to be immutable. And that's so that they can be hashed.

Why does it have to be immutable? Well, imagine that you used a list as a key. You'd hash it when you put the list in the hash table. But then you might mutate it, and the next time you hashed it, you'd get a different value, and so you wouldn't be able to find it again. So you need it to be a kind of object, where every time you apply the hash function, you get the same value.

I don't need to show you that this works. You'll just believe me, I'm sure. Let's look at a slightly more complicated hash function.

Here, I want to hash something that could be either an int or a string. So I first check and see if it's an int. If so, I'll set the value to be e. Then down at the bottom, I'll do the modulus operator again.

But if it's a string, I'm going to first convert e to an int. And this is basically the trick that people typically use when they're doing hashing, is they convert whatever thing they have, to some integer.

People do this with all sorts of things. For example, this is the way airport security systems today do face recognition. They hash every picture of a face to an integer, and then look it up.

So we can see it here. The way I'm doing it-- and the details don't very much matter-- is I'm going to do it a character at a time, do a shift ord of C, takes the ASCII, the internal representation bits of each character. And again, I don't care if you understand how the code works here.

What I just want you to see is that it's not very long. And in, fact it's typically fairly simple to hash almost any kind of an object. Because deep down, we know that every object is represented by some string of bits in the computer's memory. And we can always convert a string of bits to an integer.

All right, so that's hashing, a very powerful and extremely useful technique. Yeah?

AUDIENCE: [INAUDIBLE] you're returning something, will you always find the remainder, or can you use [INAUDIBLE]?

PROFESSOR: There are many different ways of doing hash. All the hash function has to do is convert its argument into an integer in some way or another within a fixed range. It happens to be that remainder is a very simple way to do that and is often used.

There's a whole theory about hash functions. You can read about them in gory detail on Wikipedia. The math can actually be quite complicated, and there's no real need to understand it. I typically do something like dividing by a prime number, which is known to have good properties.

But, again, you can get carried away with this. And it's usually not worth the trouble, unless you're very deeply involved in something. OK? And then I've got another program that does this, but again, I don't think much would be served by running it for you.

Any questions about hashing? Yes?

AUDIENCE: So does this only work because you said [INAUDIBLE] Python treats lists in a certain way? You said other languages, you can't have this constant time list search. So how would hashing work?

PROFESSOR: So the question is that, because Python gives us constant-time looking up for lists is the key to making this work -- every programming language I know has some concept that's similar. In, say, C, it's not a list, but it's an array. And so in C, you would use array for this purpose? In Java, you would use the arrays for this purpose?

But yes, you can do this in any programming language, because every reasonable programming language has some way to create the equivalent of a list, in which you can get a particular index in constant time. So it's a universally useful technique. Good question. Anything else?

If not, we're about to abandon searching and sorting, and in fact, abandon algorithms in general for a while. Bounced right off his hands. I will not sign you to a baseball contract.

Pardon? Yes, I threw him a Butterfinger? Oh, it's terrible. Boy, your jokes are worse than mine.

I now want to move on to the last-- go back to Python, away from algorithms, away from computer science in general, and talk about the last three major linguistic concepts in Python, exceptions, classes, and then later iterators. Let's start with exceptions, because we've already seen them, and they're pretty simple.

Exceptions are everywhere in Python. And we've certainly seen plenty of them all semester already. We get them if we set some list to, say [1,2] and then I asked for a test of 12, I get an index error. This is an exception.

We've got exceptions when we've tried to convert things to incorrect types. So if I go into test, I'll get a type error. Anything that ends in the word "error" is a built-in kind of exception in Python. We've got them when we accessed, nonexistent variables, a name error. So there are a whole bunch of these things.

In each of these cases-- here, I'm just kind of playing around in Python. And it printed an error message, and it stopped. These kind of exceptions are called unhandled exceptions, where they cause the program to effectively crash. The program just stops running.

And I suspect that some of you have written programs in which this has happened. In fact, is there anybody here who has not written a program that crashed because of an unhandled exception? Good. For those of you watching on TV, no hands went up. Almost every day, I write a program that crashes because of an unhandled exception.

On the other hand, once my program is debugged-- once your program is debugged-- this should never happen. Because there are mechanisms in Python for handling exceptions. And in fact, as we'll see, it's a perfectly valid flow of control concept.

You will sometimes write programs that are intended to raise exceptions. And then you'll catch the exception and do something useful with it. The way we use exceptions is in something called a try-except block. So you write the word "try," and then you have some code, and then "except," and then some more code.

What the interpreter does is it starts executing this code. If it gets through this code without raising any kind of an exception, it jumps to the code following the except block. On the other hand, if an exception gets raised here, it immediately stops executing this code and jumps to the start of this code, the code associated with the except, and executes that. And, then when it finishes this, it again goes to the code following the except.

Just like an if, you can nest these things. It's just a control concept. It's nothing more. Let's look at an example.

So here's readVal. This is a function. It's a polymorphic function. Its first argument is of type 'type'. Remember-- and you'll hear me say this a 1,000 more times at least-- in Python, everything is an object and can be manipulated by the program. It's one of the beauties of Python. It's something that's not true in many programming languages.

So types are objects, just like ints or floats. So the first argument to readVal is a type. And then there are two strings, the request message and the error message.

It then sets a local variable, numTries, to 0. And while numTries is less than 4, it sets val equal to raw input, printing the request message. And then it tries to convert val to valType, whatever that happens to be. And if it succeeds, it returns it.

On the other hand, if during this attempt to convert it an exception is raised-- for example, the user is asked to input an integer, and they type in the letter b, which can't be converted to an int-- then a type error exception will be raised, or a value error exception. And if a value error exception is raised, it prints the error message, increases numTries by 1, and goes back to the top of the y loop. If it goes through this y loop too many times, it leaves, raising the type error with the message that the argument "numTries exceeded."

All right. Let's run this and see what happens. Every once in a while, funny things happen. When that happens, somehow, clicking in this window, and then clicking back in that window tends to make it work again. This is a bug in Python.

So it's now asked me to enter an int. If I enter an int, everything is good. It just prints it. On the other hand, if I run it, and I enter the letter A, it will give me another chance. And now I can enter an int, and it's happy.

All right. Now, suppose I am unhappy with the fact that if four times in a row I've failed to enter an int, something bad happens. It comes back, and it raises an exception. Well, I can catch that in the code that calls readVal.

So here at the top level, I'm going to try readVal. And then I'm going to say, except if a type error is raised, print the argument returned by that exception. So this is what's called the handler for the exception. And so I don't have to crash when the exception is raised. But I can actually deal with it and do something sensible.

If following the word "except," as you see over there, I have not listed any exception names, then I'll go to the except clause for all exceptions. Doesn't matter what the exception is. I will go there. So I can write code that captures any exception.

OK. Usually, that's not as good a thing, because it shows that I did not anticipate what the exception might be. But you can see that this is a pretty powerful programming paradigm. I can write this fairly compact readVal function that's pretty robust. It's polymorphic, it can take in what the error messages are, and it can try as many times as I want. Yeah?

AUDIENCE: [INAUDIBLE] after type error [INAUDIBLE]?

AUDIENCE: PROFESSOR: You'll note that when I raised the exception type error after it, I had an open [UNINTELLIGIBLE] a string. So what this basically says is that an exception can have associated with it a set of arguments, a sequence of arguments. And I've just chosen to call the first of the arguments s, so that I could then print it.

So this is a fairly common paradigm, that you associate a message with an exception, explaining the exception. Since after all, type error is not all that meaningful. And this tells me why it was raised, that I tried too many times. OK, does it make sense? Anything else? Yeah?

AUDIENCE: [INAUDIBLE] before you said [INAUDIBLE]?

PROFESSOR: Well, what we'll see when I execute something like assert false, it's actually raising an exception. It's raising an assertion error exception. And so I can actually catch that and do something with it. I've been using these asserts just basically to stop the program. But we've also seen that sometimes in functions, we use assertions to check the types of the arguments.

But if you don't catch that exception, then the program crashes with the wrong arguments, without doing anything useful. Because that's just an exception, I can catch it, and then do something useful. So it's, again, just an example of that.

We're going to try your hands once more. I'll throw you a different kind of candy. And this time he caught it. Two for two.

Very powerful. A very useful mechanism. You can use them for sort of catching errors, as we've done here. They are frequently used in situations where you are getting input from a user.

So for example, if you were writing a text editor, and you wanted to open up a file, and you typed in the file name, and it didn't exist, you would in Python get an error message when you try to open that file. It would raise an exception, "File Not Found," basically, which you could then catch, and then print to the user a useful error message saying, "File Not Found."

Similarly, if you try and write to a file that already exists, you can get an exception saying, do you really want to overwrite this file? And ask the user. So it's very commonly used in a lot of programs as a mechanism. And now, as we go on and see more and more code, you'll see that I'm going to start using exceptions fairly frequently as a flow of control mechanism. It makes certain kinds of code easier to write.

All right. That's all I have to say about exceptions. Simple but useful mechanism. Now, on to something that's even more useful, but not so simple. It is probably the distinguishing thing, not only in Python, but in a whole class of modern programming languages. And that's the notion of a class.

I'm not going to finish it today. I'm barely going to scratch the surface of it today. But we're going to start leading up and I will pretty much finish it on Thursday.

We've already seen the notion of a module, which has been a collection of related functions. So, for example, we've seen code that includes something like import math. And that provided me with access to functions like math.log.

What the module mechanism does in the import mechanism, it makes it convenient to import a lot of related things at once. And then we use this dot notation to disambiguate, to tell us, well, which log. Typically, there's probably only one log function. But certainly, you might imagine that set.member and table.member would be different, and that both might exist. And as we've said before, the dot notation avoids conflicts by disambiguating.

OK. That's a module. What a class is, it is like a module, but it's not just a collection of functions. A class is a collection of data and functions, functions that operate on that data. They are bound together, so that you can pass an object from one part of a program to another. And the part of the program to which you pass it automatically gets access to the functions associated with that type of object.

And this is really the key to what people call object-oriented programming, a very popular buzzword. So we've already seen that kind of thing, where if we pass a list from one function to another, we can write something like L.append, some value. The data and functions associated with an object are called that object's attributes.

So you can think of this as a way to associate attributes with objects.

Now, I've been talking about data and functions as if they're different kinds of things. In fact, they're not really, because they're just objects. In Python, everything is an object, including, as we'll see on Thursday, the class itself is an object.

When people talk about objects and object-oriented programming, they often use a message passing metaphor. And I want to emphasize it's nothing more than a metaphor. And I almost hesitate to bring it up, because it makes it all sound more complicated than it is. But you will see this phrase in the literature. People will use it. So you need to know what it is.

The basic metaphor is that when I write something like L.append, I am passing the message append e to the object L. And then there's a mechanism for looking up what that object means, what that message means. And then the object, L, executes that message, and does something.

So, for example, if I had a class called Circle, I might pass the object C -- the message area, which would cause this object to return the area of its own area, the area of the circle that is that object. Again, nothing dramatic going on here. If you just think of this as a fancy way of writing functions, you'll be absolutely correct. But I did think you should hear about this metaphor.

I add by the way -- should have-- I've been using the word... Method is a function associated with an object. So in this case, the method area is associated with the object C. And purists would say, always refer to append as a method, rather than a function, because we use the dot notation to get to it, and it's always associated with some object of type list.

Now, just as data can have types, objects, as we know, have types. What a class is is it's a collection of objects with identical characteristics that form a type. So we can use classes to introduce new types into the programming environment.

So as you think about existing things-- now, I won't put that up, because it will disappear behind the screen. We've looked at things like lists and dict. What these are are built-in classes. They happen to be classes that are so useful that somebody decided they should be part of the language, they should have efficient implementations, they should be built-in, people shouldn't have to reimplement them themselves.

And in fact, there are a whole bunch of interesting built-in classes in Python. And there are a whole bunch of libraries of classes you can bring in-- and we'll look at many of those-- that extend it.

And that's the beauty of the class mechanism. It lets you add new types to the language that are every bit as easy to use as the built-in types. So in effect, the language can be extended to add new and useful types. And we'll look at several examples of that starting on Thursday.