[SQUEAKING] [RUSTLING] [CLICKING]

ANA BELL: So welcome to the last class. Please don't come on Wednesday. I will not be here. Today, we will just be tying up some loose ends regarding some topics that we've seen throughout the course. And then I'm going to do just a wrap up of things we've learned and potential courses that you might want to take after this.

OK. So today, as I mentioned, we're going to tie up some loose ends regarding lists, dictionaries. So those two topics are going to be combined into one part of this lecture. It's going to also include a little bit about complexity-- so just some things that we've learned, demystifying some details that I skipped throughout the past few lectures.

And then we're going to talk about simulations. So simulations are very, very useful-- is a very useful idea that you can already do with what you've learned in this class. And I'll show you some useful places where you can apply computation and simulation to do some interesting things. And then we'll do the wrap up.

So let's first start talking about lists. So lists were the first data structure that we encountered. That was really useful, right? We did see strings. And we did see tuples and things like that. But once we saw lists, it opened up a whole new world of possibilities for how we can manipulate data. So lists are sequences of objects.

I kind of skipped past how they're actually implemented in memory. So I do want to talk about that a little bit. But what we did talk about was the complexity, the asymptotic complexity of list operations. So some of these were pretty obvious. So the ones that are theta of n down here were obvious because, well, to check for equality between two lists you, of course, have to look at each element in the list.

So that's theta of the length of the list. To check whether an item is in a list or to iterate over a list, obviously, it's theta of n because you have to look at each element in the list. But we didn't really talk about the complexities up here. So accessing an item in the list specifically is theta of 1.

So that means, if you have a list with a whole bunch of elements in it, to grab the element at a specific memory location, it's constant time complexity. So it basically doesn't depend on the length of the list. It's instant. So we're going to see why that is.

Let's first, for simplicity's sake, assume that we're storing a list in memory of just integers. So I know lists can store other lists, and dictionaries, and things like that. But just for this first slide, let's assume all we're doing is storing integers. So the way Python does this is when you create a list-- let's say you initially populate it with length L.

Python initially allocates a contiguous memory block with length L memory locations. So if you have a list with 100 elements in it, initially, populated with 100 elements in it, Python will initially create for you a sequence of memory locations that are reserved for this list.

Then it says, well, if this is going to contain just integers, I'm going to say each one of these memory locations will hold four bytes for that integer. That's how we represent an integer. And it could be eight bytes, something else for different machines. But in this particular example, let's just say each one of those memory locations will store an integer. And that's four bytes long.

Well, if this list is contiguous-- a bunch of blocks of memory all in order-- then to access the i-th element, all you need to do is a little bit of math. So here, I've got an integer in one position in my contiguous block. Then I have maybe another integer at the next position, and so on, and so on until I have another integer at the i-th position.

So since these are consecutive, to access the location of the element in this i-th spot, all I need to do is look up that many memory locations from the start of my list. So that's just pure math. So one byte is eight bits. So if I have 4 times 8 bits multiplied by i plus the first location, that will tell me exactly the location of the i-th integer. So this is all made possible because these memory locations are allocated in order. If they were allocated not in order, then maybe this would not be as easy. Yeah?

STUDENT: Why is it 32?

ANA BELL: 32 because-- so if I say an integer is stored as four bytes, in bits that's 4 times 8 because eight bits is in one byte. So 8 times 4 is 32 bits for one byte, yeah.

All right, but this is assuming that I'm storing integers. And obviously, lists can contain other lists. They can contain tuples. They can contain dictionaries. And some of those objects might not fit within this set number of bytes, within four bytes, because some of those objects might be very, very large themselves.

So in that particular case-- let's say the list is heterogeneous-- that doesn't faze us because we can say, well, instead of storing the object itself at each memory location-- that worked for integers, but might not work when we have to store a list of 1,000 elements at a particular memory location-- instead of storing the element itself, let's store a pointer.

And a pointer is just a number that tells you which memory location that list might be stored at or that dictionary might be stored at. So if we store a pointer at a particular memory location, then that means that this is my again contiguous memory allocated for a list of length L or something like that-- then here, I'm storing still an integer. And that integer tells Python which memory location to jump to to grab the integer that's stored there or something like that.

And here, I might have another list that I'm storing. But I'm not storing it exactly in that memory location. It's pointed to by this pointer that tells Python, again, to jump to a different memory location where that list might be contiguously stored itself. So here in this example, I'm still storing numbers. It's just that these numbers correspond to a memory location that tells Python where to go to get my element in that list.

So in terms of the computation to get the i-th element in the list, it's going to be the same. I'm still allocating, in my original list, four bytes to store my pointer-- again, just a number. And so to get the i-th location, all I need to do is tell Python to go the start of this list plus 32 times i locations down to get to that element.

So this formula here, adding the start of this memory location of the list plus 32 times i, is just math. There's nothing here that depends on the length of the list. So to grab the element at the i-th location, somewhere within here, all we're doing is some math-- an addition and a multiplication.

And since that is just-- none of that depends on the length of the list, the complexity to access the i-th element in the list is constant, just math. And we're using this idea that we know exactly how many memory locations we need to jump to get to the i-th location. Does that make sense? OK.

So that leads us to the question. Well, OK, we're storing a list of elements. And we're using the idea that a list has indices to tell us the value-- there's an element at index 0, an element at index 1, an element at index 2, and so on. So there's an order to the list. And because of that order, we're able to index an element at the i-th location in constant time.

But let's say we wanted to store a dictionary. A dictionary does not have an order to it. And what does a dictionary store? It stores a key value pair. In a list, you could think of the, quote unquote, "key" as the index-- 0, 1, 2, 3, 4, and so on-- and the value as the element at that index. But in a dictionary, the key is not ordered. The key can be anything.

So here, I've got a dictionary that maps maybe a name to a Boolean. Maybe the student is in this class, true or false. So a naive implementation of a dictionary could be to say, well, let's implement elements of the dictionary-- so a key value pair-- as a list, so just two elements. The first element in that list is my key. And the second element in my list is my value.

So here, a really naive implementation uses the list. And I've got four entries in my dictionary. The element at index 0 are all strings. The element at index 1 in each location is my value associated with that key. Well, if I were to index into this list to grab the value associated with Eric, for example, can I do that in constant time?

No, right? Because there's no numerical index here. There's no order to this set of values. It's not even in alphabetical order-- so A, then J, then E, then S. And there's no order guaranteed for dictionaries anyway. So in order to look up an item in this naive implementation of a dictionary, where you're just putting all the elements in order in a list, it's theta of n, where n is the length of our list.

And so this implementation of a dictionary doesn't work. And yet, when I showed you this slide a few lectures ago, we saw something interesting. So this is what we just "proved," quote unquote, the access time in a list is constant. But the access time in a dictionary is constant as well in the average case.

In the worst case, it is theta of n. Accessing an item in a dictionary is theta of n because, in the worst case, we might store the dictionary like this. It's just a list of all of our dictionary entries all in order. So to look up one index, we'd have to go through the entire list and check if the element at index 0 is the one we're looking for, and then grab the element at index 1 as its value.

But in the average case-- and this is what we're going to see next. In the average case, the access, the time it takes to do a lookup for a key in a dictionary is constant. It's actually theta of 1, which makes dictionaries really powerful data structures to use in a lot of situations. So why is this?

Well, it has to do with the idea of hashing. So the way that dictionaries are actually stored in memory is not as a list of a bunch of entries. We just showed that that is not feasible. That leads to a theta of n lookup time. So instead, they use something called a hash table. We briefly spoke about this.

A hash table is just like a long list. And the indices of the hash table are actually things that you look up using a hash function. So how does this work? Well, any key that you'd like to add to a dictionary actually has a hash function run on it. And this hash function takes in maybe an integer, maybe a float, maybe a string, maybe a tuple.

Any hashable object hashes it, which means it takes that object. If it's a string, it'll give us an integer. If it's a tuple, it'll give us an integer. So if it hashes it, that means it takes it in as an input and gives us back a number, an integer. And that integer is what is used as the index to look it up in the hash table, to look up the value associated with it in a hash table.

So in that sense, the lookup itself is constant time because we just showed looking up an item in a dictionary using the index is constant time. And if that hash function is also constant time, then the time it takes to look up an item in a dictionary is also constant time.

So here are some examples of the Python hash function actually being run on different objects. So up here, if I run-- it's literally a function in Python-- hash of some parameters-- so in this case, 123-- it just gives me the value back. So the hash of some number is the number itself. We can hash a string. That'll give us this particular number-- so again, an integer. The hash of a tuple also gives us some number back.

So these are all just some function running behind the scenes that takes in this hashable object and gives us back a number. That's it. Now, we can't run a hash function on a list because a list is mutable and therefore unhashable. If the object changes, then the hash function run on this object will give us a different value.

So if you actually run this on your own computer, you might get different answers. Or if you run it at different times, you might get different answers because Python adds a little bit of randomness to the hash values, just in case you want to encrypt data and things like that. But generally, you will always get some integer back if you run the hash function on an immutable object.

So then that begs the question, how big should a hash table be? So if a hash table is basically just a long list, and if I run a function on some object to give me the value of an index within that list, how big should this table be? How many indices should I have? 1,000? 1 million? 10 million? What's a good number?

Well, let's take an example of a string. So for a string, what we can do is-- and we can use my name as an example. If we want to hash my name such that every single name hashes to something unique, what we can do is the following.

So we can take each character in somebody's name. And behind the scenes, each one of these characters actually has an integer-- sorry, an ASCII code associated with it, which is something numeric. And what we can do is just convert that number to binary. So the letter capital A happens to be this binary value-- 01000001. The lowercase n is this value. The lowercase a is this value and so on.

So I've got seven different groups of eight bits here for corresponding to each letter in my name. Now, if I take those bits. And now just smush them together, concatenate them to give me one really, really big number-- so this is all going to be one really big number-- the corresponding number in base 10 is this really long thing.

And so if I do this, as long as someone's name is unique, they will end up with a unique number associated with their name. And therefore, that unique number can be used as a unique index into a really big hash table. So let's think about hashing the names of MIT's 4,000 undergrads.

Let's assume that the longest name is 20 characters long. So there's going to be 20 of these different letters that we need to hash. So we use the same process here. Each one of those 20 characters gets its own 8-bit representation. So in total, the number of bits that I'm going to use to represent that 20-long character is going to be 8 times 20, so 160 different bits.

That's a lot of bits. And if I concatenate all those together, the number that corresponds to is 2 to the 160, which is this thing here. So if I want every single combination of letters in the alphabet to be a unique value in this long list, then I will need to have a list that is this long. I'm not even going to try to figure out or to say how big this number is, but it's really, really big.

And having a list, a.k.a. hash table, that has this many entries will guarantee for me that names that are 20 characters long will each hash to something unique. But I only have 4,000 names that I'd like to put in my table. So I have 4,000 names that I'd like to put in a table that has this many spots. So that's a lot of wasted space. Yeah?

STUDENT: Sorry, is it the 160 because you put it to binary?

ANA BELL: Yeah, exactly. So each one of the characters has eight bits associated with it. So there's going to be 160 of these zeros or ones in a row. So to tell the number that that's associated with, we basically say-- we basically calculate 10000000-- with 20 zeros at the end. And that gives me 2 to the 160. That's going to be how big my number is.

STUDENT: [INAUDIBLE]?

ANA BELL: That is going to be-- that's going to be for one-- that's going to be how many slots I'll need in order to have unique combinations of letters be mapped to one slot. So 0001 will map to one thing. 00010 will map to another thing. 0011 will map to another thing. So all these combinations of letters will each map to something unique.

And in order to guarantee that, I need this many slots. Again, since, I only have 4,000 undergrads, well, that's a lot of wasted space. I'm only using 4,000 of these slots to hold students' names. And that's because a lot of those combinations of letters aren't really valid.

So what's the solution? There's a lot of wasted space there. So the solution would be to say, well, you know what? I would be fine with having a smaller hash table. I don't need that giant number of entries in my hash table. I would be fine with maybe having 10,000 spots, and then having some names that happen to hash to the same thing or saying, I'm fine with having a hash table that has a million spots. And out of those 4,000, some will be used, some will be unused, and some might collide to the same hash value. And that's totally OK.

So if we allow collisions, what is this going to look like? So here's a visualization of our hash table. So think of the hash table like a list. The reason why we think of it as a list is because indexing into a list is constant time. We're taking advantage of the idea that if we index into a list, that's going to be constant.

So let's say we're adding some names and grades into our hash table. So this is our representation of a dictionary. The values here says that I have a hash table that has 16 different entries, 0 to 15. And 0 to 15 corresponds to the list index. So if I have a name and a grade that I'd like to add to my hash table, I need to run a hash function on the key. So the key is the name. And the grade is the value associated with the name.

So to add Ana with a grade of C to my hash table, I need to take my Ana, which is the key, and run a hash function on it such that, when I run the hash function on this name, A-n-a, it'll give me a number, an integer between 0 and 15. And if I can do that, then I know I've added my entry here into one of these buckets.

So a reasonable hash function to run on the name-- and we saw this in the dictionary lecture-- is to say, well, let's have A map to 1, B map to 2, C map to 3, and so on. So for my name, I've got 1 plus 14 plus 1 equals 16. But since I want to ensure that this hash function gives me a number between 0 and 15, let's mod that with 16.

So I can sum all the letters in my name just fine. And then let's finalize it by saying mod 16 to give me the remainder-- either 0 1, 2, or 15. And if I do that, I'm ensured that this key value pair will be added to one of these buckets from 0 to 15. So in this particular case, Ana with a grade of C maps to 0. That's what the hash function on my name told me to add-- the location that the hash function on my name told me to add to. So there I am putting my name in there.

Let's add a couple more people. So here's Eric. His name hashes to 35 mod 16. So that's 3. So I'm going to add Eric and his grade to bucket number 3. Then we can add John with a grade of B. His name hashes to 47 mod 16. So that's 15. So we can add John down in bucket 15.

And then let's add Eve with a grade of B. So she hashes to 32 mod 16, which is also 0. And you know what? Anna was already in the bucket 0. But that's fine because you know what? I have four names here, so four entries that I want to add to my hash table dictionary. And two of them collided. That's fine. I still have many other buckets that are empty here.

So if I have 10 students in my class, probably they won't all hash to 0. They'll probably hash somewhere within here so that it's nicely balanced. And so maybe out of 10 students in my class, only two collided. And that's way better than having all of the students in the class be enumerated in one long list.

So when I look up Ana, the way that this works is you hash the name Ana again. So when you want to look up the grade of Ana, that's the key. You hash the value Ana again. You say, hey, Ana hashed to 0. So then I'm going to look in bucket 0 and say, all right, let me enumerate everybody who's in bucket 0 and see if I can find Ana with her grade. Happens to be the first one. But if it was later on, then I'd still be able to grab it much faster than if I had everybody in one long list. Does that make sense? Like the idea of-- yeah, go ahead.

STUDENT: So you can still like access them. It's just that you might get two answers instead of one.

ANA BELL: Exactly, yeah. You can still access them. You just might have to look through a list of two. So here at bucket 0, I'm effectively storing a list of everything that hashed to a 0, which is it's fine. Yes, that's two that I have to look through. It's not four. It's not 10. It's not 100. It's not everybody all in a row. Yeah?

STUDENT: Is the complexity that theta n?

ANA BELL: So the complexity of this is actually going to be smaller than theta of n. And it'll depend on the hash function that we use. This hash function needs to be nicely balanced. It shouldn't put everyone in bucket 0. Then that's a useless hash function. And it depends on the size of the hash table.

If I have maybe 1,000 people that I'm storing in 15 buckets, I'm going to have a lot of collisions. But if I'm only storing these four, or maybe eight, or 10, or something smaller than the size of my table, then there will be far fewer collisions. It'll be more nicely balanced, yeah.

STUDENT: I meant like theta n like the things in 0.

ANA BELL: Oh, theta of n for the things in 0, yes. And that's fine because usually what we care about is theta of the length of the input. So in this case, it's theta of-- if I have four students in my class, I've got only two that mapped to 0. So here, it's length over 2. But if I had more students, then it would be far fewer. It would be 2 out of 10 or maybe 2 out of 15 that hash to the same thing, yeah.

Yeah, so as the question said, what makes a good hash table and hash function pair? Because this only works if you have a really good hash function and a nice hash table to go along with it. So this is actually a problem in computer science, a research problem all by itself. So people actually study this for their lives, coming up with good hash functions and hash tables.

So some base rules-- you want to have the domain of interest-- so in this particular case, a tuple, or a string, or whatever it is-- mapped to integers between 0 and the size of the hash table. So in the previous example, we don't want to have a hash function that mods 2 because then everything will either hash to 0 or 1. If our hash table has 15 things, well, we better make sure that our hash function is going to give us a number between 0 and 15.

Second, you want the hash value to be fully determined by the value being hashed. So in this case, we don't want any sort of randomness to go on for the reason that, well, if I want to look up Eve's grade later on in the code or whatever, then I need to run the exact same hash function on her name to determine the grade. So if there's randomness involved in this hash function, then you might not get back the same value that it has to originally. So you'll be looking in the wrong bucket. And you'll incorrectly say she doesn't have a value or she's not there.

Third, you want to use the whole input to hash the-- the function should use the whole input to run the hash function. So again, in this example, we don't just want to use the first letters of people's names because then that will lead to a lot more collisions than if we used the sum of all the letters in the alphabet-- or all the letters in their name.

So those are really big ideas. And then what we want out of our hash function is all the values. If you run this hash function on a bunch of different inputs where you're storing names or you're storing-- I don't know-- tuples or whatever you're storing, you want this function to give you a nice uniform distribution of values. So in our hash table previously here, if I add more names to my hash table, I want to ensure that they're going to land in buckets 2, 5, 6, 7, 8, 9, 10 11, 12, 13, 14. I don't want everything to hash to number 0. That would be very bad.

So as a side reminder, back in the lecture on dictionaries, I actually said something like, for now, think of the objects that can be keys to a dictionary as immutable objects. And I said, technically hashable, but we don't need to know what that is. Well, hashable means just this-- you can run a hash function on the object and you'll get the same value back no matter how many times you run the hash function on that object.

So we looked at this example. What happens if we add a student whose name is not immutable, not hashable? So lists are mutable objects. So as such, they are not hashable. That means, if we run a hash function on a list today, and then we potentially mutate the list, tomorrow that list will not hash to the same thing.

So we saw this example. Let's say Kate with a K is added to our hash table. So her name currently hashes to 37 mod 16, which is a 5. So we added her there. Now, let's say tomorrow we want to look up her grades to do whatever to integrate it into a bigger spreadsheet. She had changed her name between yesterday and today.

Now, she's Cate with a C. If we run the same hash function again on her name, that leads us to look in a different bucket. She's still there. She's Kate with a K, as we had originally added her. But now, her name is Cate with a C. We run the same hash function. It tells us to look bucket 13. And she's not there. So that's why we only want hashable objects to be added-- to be keys to the dictionary because we want the same value to come back to us when we run the hash function on them.

OK. So now, we can see, in the worst case scenario, everything maps to the same bucket in my hash table, my list. Every single thing I add has a really bad hash function on it. Let's say the hash function always returns 3. If my hash function always returns 3 no matter what I'm adding to my dictionary-- no matter what I'm hashing, then every single item essentially gets put in a really long list at bucket number 3.

So when I look up a value, well, surprise, it hashes to 3. And now, I need to look through every single thing in that bucket number 3 to find the one I'm looking for. So it's just very, very bad. And in the worst case scenario, this is the complexity. It's theta of n, where n is the length of whatever items we have that we're adding to our buckets-- to our hash table.

But in the average case-- and this is only when we have a hash table that's pretty big relative to the things that we're adding to it and when we have a hash function that's good enough, that has a nice uniform distribution of values-- only in that case, in the average case, the time it takes for us to grab a value from a dictionary is theta of 1, constant.

And so that's why dictionaries are really, really useful data structures to store things and to retrieve things from. Back when I was doing a little project, I didn't know about Python dictionaries. I had just learned about the language. And I was actually using lists to read in genomic data files. And I was storing everything in lists-- genomic names and things like that.

And it was really slow. My advisor would be like, is your code done yet? I'm like no, it's been a couple of days. It's still waiting. And then someone told me, hey, just use a dictionary to store the values. And then the lookup is going to be a lot faster. It was done within a couple of seconds.

So very, very useful, the time complexity-- because genomic data, it's huge amounts of data. So the theta of n versus theta of 1 is really-- it makes a really big difference when you deal with large data sets. It's not just on paper. It actually makes a big difference. OK. Questions about this? I hope this ties in-- yeah?

STUDENT: So Python uses a specific hash function?

ANA BELL: Yeah.

STUDENT: If Python updates or anything, Can they change that function and nothing would change? Or is it--

ANA BELL: Yeah, so Python right now uses a specific hash function. In a future version, they might use a different hash function. We don't really use the numbers associated with the hash functions. I mean, you could for your programs. But it would be, I guess, relative to whatever value you get, right? So you wouldn't hard-code the value for the tuple 123 as something, right? You just get what you get. And that's what it is, right? But it could give you a different hash function. If you ran out of your computer, actually, you might get a different hash value than mine, yeah.

So this topic kind of ties in data structures-- we've seen lists and dictionaries, some of the behind the scenes look at how things are stored, puts a little complexity in there talking about algorithms and runtimes. So it ties in a bunch of the topics that we've seen in this class really, really nicely.

So one other thing that, I'd like to now talk about is the idea of a simulation. And this hopefully is going to be a little bit more useful to you if you decide to take another computer science course or computation course in a different field, whatever you'd like.

Computation simulations are very useful tools in computer science. So it allows you to computationally describe the world. So if you see an event in the world, you can actually simulate it computationally. With what you've learned so far, you can totally simulate a whole bunch of things. And we're using randomness to simulate these events that you might see in the real world.

So for example, you might have seen the hurricane paths, when you see on the news or whatever the most likely path that a hurricane might take. But then they also have the little models that show other likely paths. They simulated using a bunch of data that they have the most likely path for that hurricane.

Another place where simulation is useful is if you see a real world event that's actually kind of complex. You can take a simpler set of rules, and simulate those, and then add in more rules to make it closer and closer to the thing that you actually observed in the real world.

So the idea of a simulation is that you have some event in the real world. And you want to calculate something about it. We're going to use computation to design an experiment. And we're going to use randomness for that. Once we've done that, we're going to repeat the experiment a whole bunch of times computationally. And that just means we're going to put a for loop around whatever experiment we've designed computationally.

And if you're interested in some outcome, some particular outcome, like-- as we're going to see, we're going to roll a die. And we're interested in how many times a 4 comes up. Then we're going to keep track of that outcome. And you keep track of it however many times that outcome happened in your whole bunch of repetitions. And then after the end of the repetitions, you can report some value of interest-- maybe the probability that a 4 comes up on a die roll.

So here's the example. It's going to be very simple because it's something we can calculate already right off the bat. But it'll give you a sense of how you can write code around such a real world event. So here, we're interested in just rolling a dice and seeing the probability to get a dot, dot, dot, dot, to get a 4 on the dice, on one of the dice rolls or the probability to get a dot, whichever.

So here, the event is that we're rolling a dice. And then we're interested in getting the probability of some face. So we're going to design an experiment for that dice roll. And this is just one way to design the experiment. There are a whole other many, many other ways to design it. This is just one that I chose that felt illustrated most how we can take a real world example and put it into code.

So a die has six faces. So what I have done here is I've created a list of each one of those faces. You could have used numbers as the elements in the list. In this case, I just used strings to be a little bit cuter, but whatever. However you'd like to represent each one of those die faces, here's a list of six things in it.

And then I'm using this choice function from this random library. Again, the random library is a super duper useful library. random.choice will effectively select one of the elements in this list for me. So if I type in random.choice in the console now, it might give me the dot dot. If I type it in right after, it might give me the dot, dot, dot, dot, whatever. It's going to be random each time I run this function.

But this line of code effectively simulates me taking a dice and rolling it. And then we can repeat this experiment a whole lot of times. If I'm taking a dice and rolling it, that's like what? One or two seconds per roll? I don't think I have time to repeat that experiment 10,000 times. But with simulation, with computation, with programming, we can simulate it 10,000 times or a million times, and then just wait a couple seconds. So very, very useful application of programming.

So how do you simulate this dice roll 10,000 times? Just slap a for loop around that line of code. So for some number in range 10,000, that means I'm going to run this line of code 10,000 times. All of a sudden, I've just rolled a dice 10,000 times. As I'm doing so, I'm interested in the outcome of some event. So let's say, how many times did a dot, dot, dot, dot come up, a 4?

Well, each time in my for loop, I can keep track of the value of roll. If it was a 4, increment a counter. If it was not a 4, I don't care, do nothing. So each time I have a counter that tells me how many times a 4 came up. And then after the for loop is done, I've repeated my experiment 10,000 times. And I can report the probability to get a 4-- so the counter divided by 10,000.

So this is the code. That's it. It's super simple. I wrote a function. And it actually takes in a parameter. So if we're interested in the probability for a dot, dot, dot, dot to come up, then we pass in the value of that particular-- of that side. If I'm interested in the probability that a dot comes up, then I pass in the dot as a string.

So what does it do? Well, just like in the previous slide, I've got this for loop here that tells Python how many times to repeat the experiment. I have the experiment number here as a variable. So I can easily just change it to be something else. And then I've got my roll here. So this is me actually doing the experiment. So just here's me rolling the dice. Here's roll value. And then I check what the value of the roll was and increment the counter if it was the side of interest, the thing I've passed in as a parameter. And then at the end, I just do a print. But you could imagine doing a return or something like that.

So if I run it, we're going to get the probability that the side dot came up was 0.167 and the probability that dot, dot, dot, dot came up was 0.1602. Intuition says they should be the same. But you know what? That's our intuition. We already know the problem. If we didn't know how to calculate the probability of one of these sides coming up, this would be pretty good.

The beauty of computation is we can just add two more zeros on there, run it, and maybe uncomment it so we actually see the values. Run it, we wait a couple seconds. But now, the probability is getting closer and closer to the true probability. So the more experiments I do, the better my answer becomes. And I just had to wait a couple of seconds. If I increase it by 10 more, I would have to wait 10 more times, 10 times as long, so maybe 20 seconds. I'm not going to do it. So it's still a guesstimate. But it's a pretty close guesstimate.

Now, the other beauty of writing code is that we can now ask, well, this is a fair die, right? Every single one of these sides comes up with an equal probability. What do you guys think the change I should make to make an unfair die? Let's say it's weighted unfairly towards the dot, the 1. Yeah?

STUDENT: You could just add the dot [INAUDIBLE].

ANA BELL: Yeah, exactly. Let me just add another dot here. Here, I've got another dot. And now, the die is weighted unfairly. It comes up more times on the 1 than on anything else. So if I run the code again, wait a couple of seconds, probability to get a 1, twice as high as probability to get a 4. So a really easy change, it helped me answer another question, a small variation of my original problem. And I didn't have to roll a dice 10,000 times in the real world.

So that was really easy simulation-- the probability of calculating sides of dies coming up is pretty simple. So why did we bother with the code? Because we could just do it mathematically. The side question that I asked was also kind of simple to figure out.

Because we can actually ask harder questions and harder variations of our original problem, we could certainly come up with mathematical solutions to these harder problems as well. But I wouldn't be as certain about my answers to those as I would be just writing code. For me, it would be a little bit easier to debug code than it would be to mathematically write probabilities to much harder questions. And you can see, once you've written the code, once you've framed your experiment in this way, it's really easy to just go ahead and change it a little bit. So the code is easy to change once it's written.

So let's look at a new question. This one says, one experiment is no longer to just roll a die once. One experiment is now that we're rolling a die seven times. And I'm interested to know the probability to get the dot, dot, dot, dot at least three times out of those seven rolls. Much harder question than before. It would require a little bit of thinking, some paper to figure out.

But in terms of code, it's going to be really simple. So now, one experiment is no longer just one choice from my list of dice faces. But it's going to be seven choices from my list of dice faces, representing the seven rolls that I have for one experiment. And out of those seven rolls, what I'm interested to do is keeping track of incrementing a counter whenever I see a four dot, dot, dot, dot.

And then, just like before, slap a for loop around it repeat that experiment 10,000 or 1 million times, however many times you'd like, and then keep track of how many times that 4 came up three or more times out of the seven rolls. So this is our event. Count how many times out of the 10,000, in that case-- but it could be a million, whatever it is-- we incremented our counter to be more than 3, more than or equal to 3. And then our value of interest is the probability of that happening. So take that counter and divide by 10,000 because that's how many times we repeated our experiment.

So this is the code. It's slightly longer. And I've actually divided it into two parts-- this one up here, and then this one down here. So the code up here is going to do the simulation 10,000 times. So I've got one for loop here that goes through 10,000 or 1 million-- however many times you want to repeat the experiment.

Within here-- sorry-- and sorry, I forgot to mention that here I have a function where I've generalized a bunch of stuff. So we could run it with different values. So instead of three times out of seven rolls, we could have 15 times out of 3,000. Right so we can generalize this.

So here, inside this for loop, I've got the simulation of rolling seven times. So here, I've got range of n roles. In the previous slide, I said it's seven. But it could be anything you'd like. And then I've got choosing one of the faces seven times and keeping track of how many times out of those seven I got a dot, dot, dot, dot.

So at the end of this for loop here, I've counted how many times I got a dot, dot, dot, dot. And then I'm going to keep track of that number in this list how many matched. So how many matched will be a list of 10,000 elements, 10,000 elements, one element for each one of my experiments.

So the first time maybe three dot, dot, dot, dots came up out of 7, then the next time 1, then the next time 5, then the next time 4, however many. So now, I've got a list of how many times the dot, dot, dot, dot came up out of 7 rolls. So the code down here-- that's why I split it up, because it's a little bit easier for me to think about it.

The code down here is now going to iterate through this list of 10,000 experiments and say which one of these is greater or equal to 3-- this one, this one, this one. So I'm incrementing a counter any time that is true. And at the end of this loop down there, I'm going to know how many times out of those 10,000 trials I had three or more times out of seven come up on the dot, dot, dot, dot.

So I can run the code. And here, I've got the exact problem on the previous slide. So if I'm interested in the probability of the 4 coming up at least three or more times out of seven roles, that's 0.0955. And then I also down here wrote it like this. And this probability is 0.16. What is this problem down here? Does it look familiar?

So the probability of a dot, dot, dot, dot coming up at least once out of one roll. That's just the previous problem on the previous slide. I just have one roll. And I count the probability to get the 4. So this matches what I got with the previous function that I wrote. But hey, now I wrote a better function actually that's more general. And I can also run it to get the probability from the previous code. So this is actually a much better code to run.

OK. Questions about this? Interesting? I mean, it's dice rolls. How interesting can it be? But yeah, so let's look at a more interesting problem, something that you might apply to the real world.

So you might see this in a calculus course or might not. But it is more of a calculus problem. So I've got water that runs through a faucet at random somewhere between 1 gallons per minute and 3 gallons per minute. This is the setup. What is the time it takes to fill the 600 gallon pool? Does anyone have an intuition for how we could solve this? If not, I can just click Next.

STUDENT: [INAUDIBLE]

ANA BELL: Yeah, definitely between the lowest rate and the highest rate, so between 200 gallons per minute-- sorry, between 200 minutes and 600 minutes, 200 at best if the water flows at 3 gallons per minute and 600 minutes at worst if the water flows super slowly, 1 gallon per minute.

So we could say, well, let's take the average of the flow. The average between 1 and 3 gallons is 2. So if we take 600 gallons divided by 2 gallons per minute, that would give us 300 minutes. It's a reasonable guess. But that's not actually the right answer.

Another way we could say is, well, let's take the slowest and the fastest it could run, it could take. So here, I've got 600 minutes and 200 minutes and average those numbers out, divide by 2. So that's 400 minutes. But that's actually not right either. Yeah?

STUDENT: Could you take the integral of 600 over [INAUDIBLE] and have the average value?

ANA BELL: You could, yeah. I don't want to do integrals, though. Yeah, but that's exactly the right answer. Yeah, you have to do an integral. Yeah, we're teaching computer science here. So we're not going to do integrals in this class. Instead, we're going to do code. And the code is going to be like five lines to find the answer to this.

So all we're going to do is grab a whole bunch of numbers between 1 and 3, a million of them if you want. These will represent a bunch of random values you could have the water flow rate be. And then we're going to say, for each one of these numbers chosen at random, how long would it take to fill the pool?

So you do 600 divided by that rate, just how long it takes. And then we're going to average all of these rates. So we have a million of these numbers, potential times that it could take to fill the pool, sum them all, average them. This is the code. It looks like a lot. But down here, the bottom half of this is just us reporting the results. Here's two print statements. And here, I'm actually also plotting what the dots look like, all the flow rates.

The actual code to do the simulation is these-- OK, I lied seven lines of code, not five. So I've got a function-- fill_pool. It can take in a size parameter. We could even do a lower range and an upper range, if we wanted to, for the flow rate. For now, we'll just hard code it to be between 1 and 3.

I've got two lists that I'm going to populate with a bunch of different random numbers. So the flow rate will be chosen between 1 and 3. So here, I've got random.random is another useful random function from the random library that gives me a number between 0 and 1. So to get a number between 1 and 3 at random, I'll just multiply by 2 and add 1. So bottom case, it'll be 1. Top case will be 2 times 1 plus 1, 3.

So r, all we need to know, will be a random number between 1 and 3, a float. So it could be anything. Append that random number to our list of flow rates. And then using that flow rate that we just randomly chose, figure out how long it takes to fill the pool, the pool of size size, so size divided by the rate. Just very simple math.

And then we now have a list that's populated with all of these times that it takes to fill the pool. And the stuff inside the loop for loop here is one experiment. So I grabbed one random number. I figured out how long it takes for me to fill my pool. And then I repeated that 10,000 times.

So down here, I'm going to report the average flow rate, which should be 2 because if we're choosing a random number between 1 and 3, the average of those numbers better be 2. And then I'm reporting the thing that we're actually interested in, which is the average fill time, which was not either of those two things we had the intuition for. But it is the integral.

And down here, I'm doing some plots. So these are the things that I've plotted. So on the left side, I've got, on the x-axis-- apologies, I forgot to label my axes and put a title on this. So I'm just going to talk about it. So the x-axis is numbers 0 through 10,000 representing each basically-- 0, 1, 2, 3, 4 represents one of my experiments, choosing a random number. And the y-axis is the random number that was chosen.

So this looks like a nice smattering of randomness here, which is what we wanted. And then for each one of these values, I'm going to have a corresponding fill rate. So for example here, if, at point 0, the fill rate happened to be 1, then that means the time it took for me to fill the pool should be up there at about 600. It could be that little point right there maybe.

So this is a graph of random numbers between 1 and 3, 10,000 of them chosen. And this is the graph of the corresponding times it took for me to fill the pool with each one of these dots that we randomly chose. We notice that the plot on the right is not uniformly scattered. In fact, it's more densely populated down towards the bottom. So are two guesses were that the fill rate was either 300 or 400 on average. And neither of those were right.

Let's view these graphs in a slightly different way. I'm actually going to the values. So right now, it was just a random number gotten between 1 and 3. But I can sort them. It doesn't matter the order that I got these random numbers. I can sort them. And if I sort them, I get something that looks like this.

So again, I've got randomly chosen numbers, 10,000 of them. And with equal probability-- that's why we see this nice line-- I chose a number between 1 and 3. Does that make sense? OK. And so then the corresponding time it took for me to fill my pool for each one of these numbers is a number between 200 and 600, as we had guessed.

Now, the average of the time it takes-- of the fill rate is 2, which is true. That is not a surprise for us because it's a random number between 1 and 3. But the actual average time it takes to fill my pool is down here. If I were to average every single one of these values, it's down here at around 330. So it's not 300. It's not 400. It's definitely between those two, but it's not really close to one or the other. That's because I've got more points more densely populated down towards the bottom than the top.

So the actual values that I got for 10,000 different randomly chosen numbers is 331. I think the actual value, if we do the integral, is like 329-point something or other. So we're pretty close. But then again, we only did 10,000. We could do a million. And we would get pretty close to the actual value.

So it's not 300 or 400. And that's because, as was mentioned from one of the fellow students, there is an inverse relationship between the fill time and the pool rate. So it's the size of the pool divided by the rate. So what we actually need to do, if we want to figure out the value, is to solve the integral between 1 and 3 of dx over x or whatever that would be.

So I don't want to bother with that. But I will bother with seven lines of code, and then just wait five seconds for that code to repeat 5 million times, or a million times, or 10 million times. Is that cool? And this is totally within your reach. It's not hard code to do. It's just for loops. It's using a random library to just randomly grab a whole bunch of numbers, and then just knowing the problem at hand-- filling a pool at a certain rate, simple math. And then you get a nice answer, a nice approximation, but still an answer to the question.

So hopefully, this shows you how powerful computation can be. This is just another example of how you can just apply computation programming to a problem that you see in real life. If you choose to do a Europe or take future courses in a different field, maybe not in CS, you will probably apply computation to concepts and ideas and problems in those fields. And it'll be something similar to this. You observe something. You try to think of it computationally, model it with a bunch of randomness, and then calculate something of interest.

OK. So that's the end of 6.100L. I've got a little wrap-up to remind ourselves where we've been and where we will go. So what did we learn? Oh, also, these slides will be like a meme dump of every remaining meme that I have. So this is going to be very meme-heavy.

So what did we learn? We've got, of course, we learned Python, programming. We learned a lot of Python syntax. Some lectures were heavier on Python syntax than others. But hopefully, you've got a handle on that. We learned, of course, flow of control with branches, if statements, [INAUDIBLE]. Else statements allow us to either do one thing or another in the code, make a decision point.

Loops-- for loops and while loops, as well as exceptions as a way for us to deal with unexpected things coming up in the course. The ideas here, flow of control, are actually transferable to any other programming language. So if you know the ideas, if you have the intuition for what kind of flow of control you'd like to use, all you'd have to do is change the syntax. And then suddenly, you can write some code in another language.

Of course, we learned about data structures. So we did lists, really useful data structures; dictionaries, super useful data structures; tuples, things like that. So you can learn about more advanced data structures in a future course if you'd like. But those are really the top two or three most useful data structures.

We talked a lot-- actually, it was a comment-- we didn't talk a lot specifically. But it was a common theme in this class, organization of code. So these ideas of decomposition and abstraction, they came up a lot when we talked about functions. That's our first introduction.

So functions helped us take some functional piece of code, some code that does something, abstract it away into a nice little module, decompose it into one little module. You have to write it once, maybe write it a few times, but debug it a few times, and then use it a whole bunch of times without worrying that it's going to change. So it's just a very nice way for us to decompose functional pieces of code.

And then we saw it come up again when we did classes, object-oriented programming. We were able to bundle behaviors and data together into one nice little object that we could then create many of in many different parts of the code, and then manipulate individually.

Another common theme throughout this class was algorithms. So we talked about bisection search algorithm way at the beginning of the lectures. We talked about it in P set 1. And it came up again towards the end when we talked about complexity and searching and sorting algorithms, things like that.

So that was kind of your only big algorithm that you saw in this class. But it shows you just how you can implement some code in a completely different way to behave in a completely different way-- to be a lot faster with some conditions, like being sorted and things like that.

And then lastly, the last part of the class was a little bit more theory heavy. We talked about computational complexity and that big theta notation. So that gave you a sense of how to maybe design algorithms. So if you have a first crack of pseudocode on a piece of paper, you can see, well, if I need to run this code on a really large data set, it's not going to work because it's too slow.

You've got three nested loops or something like that. So it might force you to rethink the design of the algorithm sooner than having already implemented it. But if you're dealing with small data sets, maybe you wouldn't care that you've got three nested for loops or a really inefficient recursion algorithm.

So those are the big things that we learned in this class. Your experience I categorize in three different ways. So you might have been a natural. If you joined this class and immediately got logic, immediately knew how to do the problem sets, that's totally fine. I still hope you got something out of this class and you learned something.

You might have joined the class late. If you found 6.100A to be too fast or too challenging, you might have joined it joined it late, kicked it to the curb, and said, let me join 100L. I welcomed you. We did a little bit of research and found that even if you join late, it does not actually hinder your performance in the class. So hopefully, that was your experience.

Did you work hard? So maybe you didn't get all the concepts right away. Maybe you struggled a little bit with the problem sets. Maybe you struggled a little bit on the P set or on the quizzes. But I still think that you learned a lot. And the test is always to go back and look at the first problem set.

So if you do that when you go home tonight, you look back at the first problem set in this class, you look back at the code that you wrote, it will seem so easy. I promise you this. And that's because I think you all did such a good job. You tried your hardest in this class. I know it's not easy. I know it's slower paced. But it's still not an easy class. And I think you've learned a lot. Looking back at the first problem set will show you that for sure.

So what's next? There have been some questions about what are some future classes that you might want to take or what can you do once you've finished here. Here we go. So we've got 6.100B. It is the most natural next step. It's a half semester class in the second half of the semester. So they're finishing up right now basically.

It's an overview of really interesting topics in computer science and with a focus on data science, though. And I actually run that class as well. So what we talk about there is optimization algorithms. So for example, let's say you want to create a schedule for your classes next semester. And you will have some constraints. You don't want to have classes at 8:00 AM, or 9:00 AM, or 10:00 AM, or 11:00 AM. And you want it to all be within some time limit or things like that. Optimization algorithm could be something that you write. And you could just apply it to something that you have.

Simulations, exactly what we saw today of the physics filling the pool thing-- you'll see more examples of that and ask different questions about it as well. So you'll see things like calculate things like standard deviations. How many times do we need to repeat this experiment in order to be within some confidence interval? So how confident are you about your answers? So we'll be doing more things like that.

And then there's, of course, the machine learning aspect of it. So if you have a bunch of experiments that you do that you get a whole bunch of data from, how can you fit a curve to those experiments? And then for a future experiment, what's the expected value? So that's a little bit of machine learning on experimental data.

And then some more machine learning in a more classical sense is dealing with clustering algorithms, and classification of data, and things like that. So 6.100B I know a lot about because I also teach it. It's a really good next class, if you want to be employable for an internship. If you take this, I think you'll be good to go.

6.101 is also a really nice class to take next. If you really enjoyed the programming in this class, 6.101 will be your next step. It's called Fundamentals of Programming. And it is in Python. It's pretty fast paced. So there will be problem sets every week. And they're going to be pretty hardcore.

There's going to be a lot of debugging involved in those problem sets. And I actually was a recitation instructor for that class. And to get a first-- for the problem sets at least, to get something working doesn't take that long. But to make it work according to the specifications that they have will take a little bit of debugging and reworking.

That's because they deal with a lot of real world data. So writing code that's really efficient is very important. So again, writing nested for loops, of course, we can totally do that. But making it efficient using data structures like sets to make the code efficient, using proper algorithms that are efficient is a very important part of this class. But you'll learn a lot if you take this class. You'll be very employable for an internship in some computer science company.

6.102 is also a nice next class, if you're interested in software construction. It is actually in a different language. It's in TypeScript these days. It used to be in Java. You can take what you've learned here. And if you're interested in learning a different language, this is a nice one to try.

Their motto is you're writing code that is safe from bugs, easy to understand, and ready for change. So they have also lots of problem sets. But you're also working in a team. So you get to learn how to work in a team well with other students, how to code together, how to write code that has nice documentation, lots of debugging, things like that. So more of that those ideas of decomposition abstraction that we learned in this class will definitely be prominent in this class, in 6.102.

And then, of course, we've got other classes I'm happy to chat about. So machine learning is a nice one, again, if you have a handle on programming really well and want to try just applying programming to machine learning. An algorithms class is also a fine next step, if you're more interested in the complexity part of this class that we saw, also very, very reasonable things to try to do next after this class.

Yes, last slide. If you're not going to code for a while, but think you might code in a couple of semesters or something like that-- you want to take a more computational class in some desired field-- I would say that you should try to practice coding at least once a week.

So in our website, we've got a little help menu where you can go to-- we've listed some other websites. There's a little bit of coding practice you can do. It doesn't need to be a lot-- 30 minutes once a week, something like that, just so you don't forget to code can go a really long way because I know, over the summer, sometimes I don't code for a month or so because I do other stuff besides coding in my life.

And coming back into it takes a little bit of time. And it's just without practice, like with anything else, it's just easy to forget. And it's hard to get back into it. So even if you just do a little bit of coding, write a really simple program once a week, it's going to go a long way.

So that's it. I want to thank you all for being in this class. And thank you for coming to this last lecture. I know you didn't have to. But I do appreciate it. Happy coding. And good luck with exams. And have a good break, everyone. Thank you.

[APPLAUSE]