[AUDIO LOGO]

ANA BELL: All right, everyone. Let's get started. So today's lecture will be on this thing called dictionaries, and it's not the dictionaries that our parents and grandparents used. Notice, I never actually used regular book dictionaries either, maybe once in my entire life.

But it's actually on a Python dictionary. So this is going to be a new data type that we have not worked with before. But it'll be a compound data type, much like we've seen lists and tuples to be. It's just going to be very different than lists and tuples.

So before I introduce a bunch of syntax and what a Python dictionary is, let's try to just motivate the need for such a data structure. So suppose we have the following problem. We've been dealing with this problem in many of our lectures. But we, once again, want to store student information.

So let's say we want to store grade information for a bunch of students. With what we know so far, we can store information using lists. It's a very reasonable data structure to use because we might get new students in the class. Students might drop. Grades might change, things like that. So let's use this mutable data structure a list.

Let's say we want to store names of students and their grades in the class, like their final letter grade. Additionally, we can store things like micro quiz grades and psets grades. But for now, let's just assume we're storing just the names and the final grades in the class. So if we do this using lists, one reasonable way to store this information is by saying, well, I'm going to have a list of all the names of the students in my class.

I'm going to have a list of all the grades of these students in the class. And I'm basically going to go index by index and make the rule that says, at a particular index, I'm storing all the information related to this one person. So at index 0 here, I'm storing the name of the student and their grade. At index 1, I'm storing the name of that student John and their grade. At index 2, I'm storing the name of that student and their grade. And at index 3, and so on and so on.

So now I basically have to remember that, for a particular index, I am storing all the information related to that student, right? OK, so seems like a reasonable way to do this. Now, let's say that I wanted to look up the grade for a particular student. So I write this function called get_grade. It takes in some parameters.

So the first thing it will take in is the name of the student, so Ana, for example. And I would pass in the list of all the names in my class and the list of all the grades in the class. So these two lists that I've previously created, right? So these get passed in to this function. So you can imagine if we have a list of everybody at MIT, these lists are going to be pretty large that we're passing in as parameters.

How do we actually grab the letter grade associated with a student? Well, we're going to use the fact that the letter grade for the student at index I in the grades list is going to be grabbing the letter grade for the student at that same index in the name list. So we have to figure out this particular student being passed in here, what index they're at in the names list.

So that's what this line of code is doing. It's using this index function on the name list with a parameter, for example, Ana. So this will return for us the index where Ana is in my list. So from the previous example, it's going to say that-- it's going to return the number 0 because Ana is stored in the name list at index 0.

So now that I have that index in hand stored in variable I, can just index into the grade list at that same index, OK? So I can get grade list at index 0 will return for me the grade that I got for that particular class or whatever we're storing here, OK? And then we just return the tuple student comma great.

So this becomes really messy, right? I already mentioned that if I have a list of a whole bunch of students for a really large class or the entire university, then it becomes really unwieldy to just keep passing in all these lists. If I have in addition all these micro quiz lists and all these problem set lists that also store additional information for the student, I then have to pass those in for their respective functions. And so it gets really messy, right, writing these functions that retrieve this information.

And additionally, if we're mutating these lists, like if a new student comes in and we need to add all their information, I need to make sure to update every single one of these lists that I'm maintaining. If a student leaves or drops the class, I need to remember to remove that index from all of these different lists. So really, really messy situation that we could get into by using this method to store information about students.

So let's try a different approach. Instead of using all of these different lists, let's say that we're going to store everything in a master list. So we're not storing many lists. We'll just store one list for the grades in the class. And the way that it will be stored according to in this slide is going to be this grades list. OK, so this is one list with three elements in it. And you can imagine if we have more students we would just put all these students in this master list.

So what is each one of these student elements? Well, each student element is itself a list. So already, I've got my master list. And each element within this list is also a list. So this is a list for Eric, a list for Ana, and a list for John. These are variable names. What are these lists going to be comprised of? Well, they will be comprised of three things.

So notice two commas here. So the first thing is their name. The second thing is another list containing their problem set grades. And I'm using this element of that list to denote what that set of numbers represents. And then another list as my third element being the scores for the micro quiz grades. And again, I'm denoting the first element of that list telling me what this list contains.

OK, so I've got lists, master list with three sublists for my three students. And each one of those lists contains three elements, a string, a list, and another list. And those two lists are then also comprised of a string and a list themselves. So super complex. A data structure-- or a sort of composition or design choice that I've made here. But it solves the problem of maintaining all these different lists in separate variables.

So now, let's say I wanted to write a function that gets the grades for a particular student for either problem set or micro quizzes grades. This is the function that does that. So again, it's not looking super nice. So what is this function going to take in? The who is going to be a string representing the name, so for example, Ana. The what will be also a string representing what information I'd like to grab, either ps or mq. And the data is going to be my master list of all the grades. So this grades equals this list of every body.

So what is this code going to do? Well, it has a for loop down here and a nested for loop inside it. The outer for loop basically looks through each one of these elements here and looks at the element at index 0, so either Eric, Ana, or John, and grabs only the list where that piece, this string here matches the who. So if student at index 0 equals who right here, then we found the student I'm interested in grabbing the information for. Cool. So now I've gotten I've grabbed the right piece, the right list.

And now I'm interested in their grades for a particular what, so either mq or ps. So I do the exact same thing again for that list here, right? So if I'm interested in Ana's ps grades, I grab these lists here. And then I'm going to check if the-- info at index 0, so either this ps or this mq matches the what, so either ps or mq to match what I'm interested in grabbing, the information, what information I'm interested in grabbing. And then I'm going to go inside this if statement if they match, and then I return the who and the info.

So again, super complex. No need to understand this that well because we're not going to use this method for long. So this get_grades here, for example, if I grab Eric's micro quiz grades and I run the code, it will return for me this tuple that returns for me the name of the student, and then just this sublist of the thing that I was interested, in this case, micro quiz.

And it grabs for me all the grades. And then I can then index into this returned tuple to grab either the first quiz or the second quiz grades. OK, and same for Ana. In this particular case, it grabs for me just the tuple with my name and then that sublist with the problem set grades.

OK, so again, really messy. I've made my design choice for how to create all these lists with sublists and sublists within those. And so I'd have to document that probably if I was using this method. And then this function to grab this information, again, super complex. Hard to read.

So it's not really a great way to store information either, but the idea behind this, which is to try to store some data associated with some sort of key, right, the ps or mq, or in this case, I'm storing a bunch of grades for Eric or Ana or John.

That idea we can explore. And that's basically what dictionaries will do for us. It will allow us to create data structures that map some sort of custom index, a key, to some value. So much like a book dictionary does, it maps a word to its definition. We'll be able to create our own dictionaries that map some object to another object.

So when we create a dictionary, we call every quote unquote "element" in the dictionary an entry. And that entry is that mapping of a key to a value. So just to draw a parallel with the list, we can think of a list as mapping something to another something. The thing that a list maps is this index, numbers 0, 1, 2, 3 in that order, right? So it has to have an element at index 0, and then that index increases by 1 from thereon, OK?

And for each one of these indices, I mapping that index to some element in my list, right? That's basically what the list does. There's something associated with index 0, something associated with index 1, and so on. So it's like a very restrictive dictionary, right?

An actual Python dictionary works in a similar way except that now I am not putting any restrictions on my indices. My indices here become these custom indices called a key. And so now I'm able to associate a value, equivalent element in my list, with that key. So I can have an element associated with any object.

So I am using the term value here, and in a dictionary the key is associated with a value. And that's one entry in the dictionary. Now, this is going to be a little bit confusing because we've been using the term value to refer to just some object's value, right, like int-- variable a has value five or something like that.

But now I'm going to try to make a conscious effort, now that we're introducing dictionary and dictionary values associated with a key, to whenever I'm talking about the dictionaries value to say dictionary value just so it's not confusing. But just keep that in mind. It can be a little bit confusing at first, now that we're using the same terminology for two different things.

So we're going to go through-- in this lecture, we're going to introduce a bunch of syntax and operations with dictionaries. And there will be lots of you-try-it exercises just to give you a little bit of practice with the syntax because this is kind of a syntax-heavy lecture. So hopefully it helps a little bit. But let's first see how to store data in a Python dictionary.

So as I mentioned, a Python dictionary stores entries. And that entry is a key value pair. So you're mapping one key to its value. The key can be any immutable object, and we're going to see what this means in a little bit. And the value associated with that key, or the Python value associated with that key, can be any object you'd like, even lists or other dictionaries.

So the way we create a Python dictionary is by using these open and closed curly braces. So tuples were open and closed parentheses. Lists were open and closed square brackets. Dictionaries are open and closed curly braces. And this creates inside memory an empty dictionary, so a dictionary with zero entries. So the length of that dictionary is 0.

To create a dictionary with one entry in it, again, we have curly braces. And we add one entry in it. So this something colon something else is an entry in my dictionary, one entry. And the thing before the colon is the key. And the thing after the colon is the value associated with that key. So you can think of it, if we're drawing a parallel to lists, this is now mapping at this custom index 4, we're putting element 16.

OK, so we can also create dictionaries that aren't just full of integers. And you can mix and match data types as you'd like. But usually, in dictionaries we have the keys all be the same type and the values all be the same type. But you can certainly mix and match types, just like you could create lists and tuples full of an integer and a float and another list and mix and match in that way.

So here I'm creating a dictionary, again, open and closed curly braces starts my dictionary. And it has four elements in it, sorry, four entries in it. And each entry is separated by a comma. I've got here my first entry. So it is mapping the key Ana to the dictionary value B. My second entry key mapped to value A. Third entry maps key John to value B. And last entry maps key Katy to value A. So this is a dictionary that essentially maps strings to other strings.

So you can see here I've visualized the dictionary that we just created. We've got these custom indices, right? So we're basically mapping names to letter grades, OK? Everything OK so far? Does it make sense, I guess, conceptually? OK, awesome.

OK, so the first thing we'd like to do is, once we have a dictionary full of a bunch of entries, how do we grab an entry? How do we look up a value associated with a key? So the way we do that is in a very similar way to the way we look up an element in a list, right? A key in a dictionary is just a custom index. So how did we look up an element in a list? So if I wanted the element at index 3, I would basically say L squared brackets 3. And that grabs for me the value at that index.

Well, now, I've got my custom indices, right? My custom indices are these strings. The syntax will be exactly the same. I've got this custom index I'd like to look up. So I say, dictionary, name, square bracket, custom index. So if I say grade square bracket John, Python will go in to my dictionary named grades. It'll look up the key John. And it'll return for me the value associated with that key, B.

So this entire expression here evaluates or gets replaced with the string B. Just like when we indexed into a list, L1 square brackets 3, we replaced that entire indexing operation with the value of the element at that location. So similar here.

If I try to index into a dictionary and that key doesn't exist, so notice my dictionary has no string Grace, Python will give me a key error. So if you run code with dictionaries and you get a key error exception being raised in the console, you will know that you're trying to index into a key that doesn't exist.

So the question might be, yes, we're able to look up a value given a key. Can we do the same thing but backwards? Given a key, sorry, given a value, like A, B, C, whatever, can we look up a key associated with that value? And the answer is no. We'd have to write some sort of loop or some sort of code that goes through every item in my dictionary to check each value and see whether the key associated with that value is equivalent to the one I'm looking for. So there is no nice expression to do that backward operation.

And that's because the values in my dictionary can be repeated. So if I look up the value B, and I want what's the key associated with B, well, there's actually two of them. So how does Python know I want both of them? How does it know I want only one of them? How does it know I want maybe a list of all these things? It doesn't, right? So you'd have to write code that does something for that operation. And we're going to see how to do that later.

OK, so let's have you work on this You Try It. And this is just an exercise in looking up a value. So this is a function I'd like to write according to the specification. So it's called find_grades. Grades is a dictionary mapping student names to grades, so string-to-string exactly like we've seen in the previous slide. And students is going to be a list of student names. So in the example here, I've got my input dictionary, this thing we just saw. And then my list of student grades is, for example, these two strings Matt and Katy.

For a bunch of these questions, especially in the micro quiz and things like that, if it gets a little confusing when I try to write the specification in a very detailed way to make it clear what I'd like from this function, it's important to try to use the example to help you figure out what we'd like because we're writing the specification in a general sense. But the example should hopefully make things really clear for what we'd like.

So in this particular case, what we want the function to return is a list of the grades for the students being passed in, right? So we look up Matt. We see that their grade is a C. We look up Katy. Their grade is an A, so I want to return the list C comma A in that same order that I passed in my students.

So I'll give you a couple of minutes to work on that, and then we can write it together. So that's line 94. So this is just an exercise on looking up values in the dictionary.

All right, does anybody have a start for me? Yes, please.

STUDENT: [INAUDIBLE]

ANA BELL: Yep, Lnew. How about that? So this will be my results list. Yep.

STUDENT: [INAUDIBLE]

ANA BELL: Yep, for loop.

STUDENT: [INAUDIBLE]

ANA BELL: Yep, so grade square bracket lm looks up the value associated with my student named lm. And maybe we can save it like this grade equals this. And then you said append? Yep, so we can do Lnew.append the grade. Anything else?

STUDENT: [INAUDIBLE]

ANA BELL: Yep, a return. So we can return Lnew. Yep, so very reasonable code. I like it a lot. Besides the first lecture, I don't know that we've written any code that didn't involve a loop. So your best bet for writing code for any sort of thing in this class is to think, what loop can I do? So let's run the code, and it should return for me C comma A. And it does.

Now that we can iterate-- so I mentioned this before, but once we're iterating over tuples and lists and things like that, one thing I would add, just for debugging purposes, is say something like lm is, and then you can say an example of what it could be, like Ana or Matt or whatever it could be, just to remind yourself that that thing, that loop variable is a string. And so it's one less thing to remember as you're writing further code. But this is really nice.

OK, so dictionaries are already proving to be really, really useful. We can create values associated with custom indices. And if we want to grab the value associated with that custom index, it's really just a matter of indexing using a key, using that specific key, much like we did indexing into a list, OK? No need to loop, none of that iteration. It's just a single line of code that indexes into the list.

So let's see a few more operations before we do the next You Try It. So I've got my list of grades that we've been working with in the past couple of slides. Let's say that we now want to add a new student and their grade. The way we do that is very similar to the way that we would add an element to a list once we already have an index for that list, right?

Here, notice we don't actually have a slot for Grace. Yet, I'd like to add her to my dictionary. That's OK. With this particular syntax here, so grades at key Grace, if Python does not find Grace in my dictionary of keys, it will just add her, OK, which is really nice, right? I don't need to check if she's already in there. There's no looping. You just say grades at Grace equals A. Boom, it adds it for you.

What if I want to change an entry in my dictionary? Well, let's say I want to change Grace's grade to a C. Grades at custom index grace equals C. We'll go in, look at my keys. When Grace didn't exist, Python added her with her value. But she already exists there, so Python will just overwrite her value.

So really nice, something to look out for in case you already have values in the dictionary. You want to be careful if you actually do want to overwrite things. But it's really, really nice behavior. And it's different than lists, right, especially adding an entry to the dictionary.

You can delete entries, much like we deleted entries from a list. We use the del function. And the del function says, what entry you'd like to delete from what list? So here we just say the name of our dictionary at index Ana. So this will completely remove Ana and her value and the value associated with Ana from the dictionary.

So what I want to make a note of is that our dictionary is being mutated with all of these different methods or all of these different functions, right? So here, when I added Grace, I've mutated my original dictionary, right? The animation didn't make a copy of this dictionary with Grace added, leaving the original unchanged. I've literally gone in and mutated my original dictionary to add Grace. I've mutated the original dictionary to change her grade. I've mutated the original dictionary to remove Ana from the dictionary, right? So all these functions are actually mutating my dictionary.

OK, one other very useful thing that you can do with dictionaries is to check if a key is in my dictionary. So we do this using in operator, this in keyword. We've seen in keyword being used to check if an element is in a list, to check if a substring or a character is in a string, to check if some element is in a tuple. We can also use it to check if an element or a key is in my dictionary.

So I want to make a note. It's only checking the keys. It does not look for the values in the dictionary. We'll see how to check if some value is in the dictionary in a little bit. But the in keyword specifically only looks at the keys in the dictionary.

So if I have the expression-- the string John is in grades, Python only looks at the keys and say, yep, there it is. I don't care what value's associated with it. I just care that it's in my keys. So this entire expression here, John in grades, will evaluate, so be replaced with true.

Daniel, obviously, is not in my dictionary keys. So it returns false. B is not in my dictionary keys, even though it's in my values. It still returns false because it only looks at the keys.

All right, let's have you try this exercise. So function is called find in L. Again, we can use the specifications and the example to help us figure out what we'd like from this function. So Ld is going to be a list of dictionaries. So in the example here I've got three dictionaries defined, and the first parameter here, the thing being passed as Ld is the list with D1, D2, D3 as my elements. And k is just an integer.

What I'd like to do is return true from the function if that k is a key in any of these dictionaries and false otherwise. So as soon as I see a key that matches k, I want to return true. So in this example here, when I look for the k 2 inside these dictionaries, D1 doesn't have it, but D2 has it. So I would return true. When I look for 25 in that same list of dictionaries, 25 is a value in one of these, in D3, but it's not a key in D1, D2 or D3. So that would return false.

All right, so that's just a little lower, line 115. Give you a couple of moments, and then we can write it together like usual.

All right, does anyone want to start me off here? So how can we do this?

STUDENT: Create a loop.

ANA BELL: Create a loop, yes. For?

STUDENT: [INAUDIBLE] for d in Ld.

ANA BELL: Yep. OK, so that means that D is-- I can say like K1 mapped to v1 or something like that, right, key to a value. If k in D? Yep, so that will check for me my keys in that particular dictionary that I'm looking at right now. Yep, we can immediately return true, right? As soon as we found it, no need to check the other dictionaries. Just pop out of the function and return true.

STUDENT: [INAUDIBLE]

ANA BELL: Same inside the if or inside the for or outside the 4.

STUDENT: Outside the 4.

ANA BELL: Outside the for, we can return false. Yep. I like this code a lot. Uses this in operator to do the task. So the return false outside of for loop works really well because if I've gone through every D inside Ld here, then I'm checking every single dictionary, right?

As soon as I find one that has that key, this return true acts like a break and a return, right? So it breaks out of the loop and returns immediately. And it doesn't return false. But if I've gone through every dictionary and didn't find the key matching k, then I return false. Yeah.

Did anybody try it a different way, or is this-- we could certainly try it with a Boolean flag, right? We could flag the fact that we found it through some loop. And keep track of it, and at the end, just return that flag. That's another way to do it. But this is probably the most Pythonic way.

So we can run it on these two examples here, right? So I'm looking up 2 to return true and looking up 25 to return false. And it does. Questions about this code or dictionary so far. Is everything OK so far? OK.

All right, a couple more operations. So, so far, we've looked up values in a dictionary. We've added stuff to the dictionary. We've deleted stuff from the dictionary. One really useful thing to do is to be able to look at every single entry in my dictionary. The reason why we'd want to do this is because we should assume that when we create our dictionaries there's no order to them, right? This is very much unlike lists. Lists had an order to them. We knew that the first element in our list was at index 0. The next one was at index 1, and so on, right?

Lists were ordered sequences of elements. But dictionaries are not ordered sequences of elements. That's not super true. Up until a very recent version of Python, there was no guaranteed order. They were put in some order that I couldn't figure out how it was determined.

But I forget which Python version, maybe 3.6 or something like that, started to guarantee an order for the dictionary elements. And that order was the same order that you inserted the elements, OK?

But if you'd like to write robust code that could be run by people using an older version of Python, you should write the code assuming that no such order exists. And it's OK. It doesn't make the code that much harder to write. But if we're not assuming any order to Python entries in the dictionary, then that means a lot of times, we actually have to look at each entry in the dictionary to do some sort of task.

So one of the first things you might want to do is to iterate through all the keys in the dictionary. To do that, we use a function called grades.keys. And this grades.keys function here doesn't mutate the dictionary at all. But instead, it returns for me an iterable, a sequence of values, which are all the keys in my dictionary.

Now, the data type of this return value is called dict_keys. It's not a data type we've worked with before, OK? It looks really weird. But if you'd like, and you don't have to do this, you can always cast this sequence of values-- that's type dict keys-- to a list, like this. So if you cast to a list grades.keys, it gives for us this more recognizable list with each key being an element in the list. You don't have to do this. But if it makes it easier for you, you can, OK?

So this line of code here, grades.keys returns for you-- you can think of it like this iterable, this list of all the keys in the dictionary. Again, they're not ordered, right? They're ordered in the order that I added them into the dictionary, right, Ana, then Matt, then John, then Katy. But they're not sorted in alphabetical order. If you have integers, they won't be sorted in ascending or descending order. So it's best to just not assume an order to begin with.

Similarly, we can get an iterable of all the values in the dictionary. And to do this, no surprise there, we use grades.values. And this is, again, a function which doesn't mutate the grades at all. But instead, it gets replaced with this dict values data type. I've never seen it before either. And you can cast it to a list if you'd like because it makes more sense to us at this point in time, which just returns for us this list of every single value in my dictionary. Again, no order, right? We can see that there's no order except for the order that we actually added the elements in. Yeah.

STUDENT: When you said like it acts like a couple, you mean like if I do grades that, at other times, it will print out the same list?

ANA BELL: Yeah, yeah it'll print out the same iterable, I guess, if you do it again, yeah.

STUDENT: If we're just doing like-- if we're just iterating over the dictionary, there can be chances where it comes up [INAUDIBLE].

ANA BELL: If you're iterating over the dictionary, not in the Python version we're using. But in a previous version, if you ran on your machine or if I ran the same code on my machine, it might have given me a different order.

But in the versions we're using from now on in Python, because you guys all probably downloaded the latest version of Anaconda and Spyder, it will guarantee the order that you inserted the elements in. But if somebody's using an older version of Python takes your code and runs it, they might actually get A, A, B, B or some other order for these functions here.

STUDENT: [INAUDIBLE]

ANA BELL: Yeah, you're welcome. So these being iterable just means that we can have something like 4 I-- so 4 k in grades.keys basically giving us a loop where k is going to be each element in this list. So that's fine. So we can iterate over the keys, or we can iterate over the values directly. But what I find personally most effective is to iterate over each entry in the dictionary. So not just over the keys or the values by themselves, to iterate over the keys and the values together.

So to do that, we use this function called grades.items. And unsurprisingly, this will return also an iterable, where each element in my iterable is not just the key or the value. It's a tuple of the key comma the value, OK?

And again, we can cast it to a list to give us something that's more recognizable. You can see now each element in the returned list is going to be the tuple where I have an entry, right? So my entry Ana comma B is this first element in my return list, and then Matt A and then John B and then Katy A.

So I grab these entries together where I have access to both the key and the value for that entry, which means-- and this is the important part-- that we can do something like this. And we can do this for the previous slide as well. But for this particular grades.items iteration, if we're grabbing a key value pair out of items, that means we can do something like this. For k comma v in grades.items means that Python will map k to the key for that entry and v to the value for that entry as I'm iterating over each one of these pairs, right?

So with each iteration, I have access to both the key and the value for that entry, which is pretty useful. So if I have this line of code here, if I print key k has value v, the k and the v will change with each entry, right? I'm just grabbing both the key and the value for that entry. Yeah, question.

STUDENT: To use [INAUDIBLE] that item, is it actually like-- is it actually tuple, or just the actual object?

ANA BELL: It's not a tuple. So the actual object type is this thing dict_items. So again, not a type that we've worked with before. But that's just the type, right? We've seen lists, tuples, dictionaries. Dict_items is another data type, yeah.

But the cool thing is that it's an iterable. So it's a sequence of values, which means that you can cast it to a list, which is also a sequence of values. And it knows how to do that casting. And you get the more recognizable list that we've been using. Other questions.

OK, so I really like using grade.items to iterate over entries. So let's have you try this exercise. So it's a function called count matches. It takes in one dictionary d. I didn't say what the elements are, but you can mix and match. So here, I have a dictionary with just int mapped to ints. And here I've got a dictionary where it maps ints and strings and things like that. And what I want this function to do is tell me how many entries in this input dictionaries have the key match its value.

So here, in this first example, the key here is 1. The value is 2. So they don't match. These don't match, and these don't match. So the count should be 0. But down here in this example, the 1 doesn't match 2, so that's fine. But the key A matches its value, 1 count. Key 5 matches its value, 2 counts. So this should return count 2. All right, let's have you work on that down by line 137. And then we'll write it together.

All right, how can I start this? Yes.

STUDENT: A count.

ANA BELL: A count, yes.

STUDENT: [INAUDIBLE]

ANA BELL: 0, yep.

STUDENT: A for loop?

ANA BELL: A for loop, yep.

STUDENT: [INAUDIBLE] v, k in d.items [INAUDIBLE].

ANA BELL: Yep, as a function, yep.

STUDENT: If v equal to k?

ANA BELL: v equal to k. Yep, so this is where my value equals my key for that particular entry.

STUDENT: Count equals count plus 1

ANA BELL: Count equals count plus 1, perfect. Yep, return count. Did anybody do it a different way? OK, awesome. Yeah.

STUDENT: Why do you have to call it like [INAUDIBLE]? I had to do it a different way [INAUDIBLE].

ANA BELL: We can write it, yeah.

STUDENT: Yeah, I did like the count equals 0, and then I did like for x in like [INAUDIBLE] that it was just going to call the value. And [INAUDIBLE] like at x equals [INAUDIBLE]. I thought that it was just going to call [INAUDIBLE].

ANA BELL: Yeah, so we can say for x and d.keys or something like that, right? Something like that, or no? We can also say for x and d, I think. That might work too because it'll grab the key for us. But just to be safe, keys. And now we need to grab the value. So how do you grab the value associated with key x?

STUDENT: Like key brackets. No, brackets are--

ANA BELL: Yeah, square brackets. It's just indexing, right? So d square brackets x. Oops, if d square brackets x equals-- so that's the value equals the key, right? Then, again, we count plus 1. So this is our other way. Yep.

So we don't have to use items, but items makes things easier because we have in hand a variable that's the value and the variable that's the key. And doing things like indexing starts to get confusing if-- it can be confusing. But yeah, both ways are very valid. So let's run it, and it should work.

So the first count is 0, as we expected. And the second count is 2. Any questions about this code? Does it make sense? Is there another way that somebody tried it? Nope? OK, good.

OK, so dictionaries are mutable objects, right? So all the aliasing and cloning rules apply. Remember when we talked about lists and using the equal sign between a list and another variable name? Just a plain old equal sign means that you are making an alias for that list.

Same thing applies to dictionaries. So saying D1 equals D2, where D2 is a dictionary, it means that you've just created an alias for that dictionary. So if you change the dictionary for either of those variables, you're changing the object itself. If you want to actually make a copy, you use d.copy where d is the name of the dictionary you'd like a copy, and that gets you a copy of that dictionary. And then you can change it without changing the original.

So let's talk a little bit about the values for a dictionary and the keys because there are some restrictions on the keys for the dictionary. No restrictions on the values. So dictionary values can be any type, right? You can have a dictionary value that's a float, int, string, tuple. You can have a dictionary value that's a list, which is immutable object. You can have a dictionary value that's another dictionary. All are OK, whatever you'd like for the values to be.

You can have dictionary values that are duplicates. So you can have one key that maps to value 5, another key that maps to value 5. All good. OK? So the values don't need to be unique. We do have restrictions on the keys though, OK? The first restriction on the keys is that it has to be unique, right? So if you're mapping a key 1 to value 5, you cannot map a key 1 to value 6 because if you go and look up the value associated with 1, how does Python know which value you'd like, the 5 or the 6, right? So the keys have to be unique, first of all.

Second, the keys have to be immutable, technically hashable. But for the purposes of this class, just think of them as having to be immutable. So a key can only be one of these types that we've seen so far, int, float, string, tuple, or bool. You cannot have a key that's a list. You cannot have a key that's a dictionary because they're mutable objects. So let's look at that a little bit further in detail.

So the reason why we can't have a key that is mutable is because of the way keys are stored in Python, sorry, the way the dictionaries are stored in Python. So I'm going to show you an example on the next slide. First I'm going to explain how they're stored, and then we'll go through an example showing you exactly why you can't have a mutable structure.

So the way dictionaries are stored in Python is you first need a key to associate with a value. So everything starts off with the key you'd like to add to your dictionary. So Python basically runs a function called a hash function on the key.

For simplicity's sake, let's say the key you're trying to store is a number. That hash function might return that same number. It might return something else. If you're trying to store a string as a key, Python again runs maybe a different hash function that takes in that string, which might be a bunch of characters, and it converts it to some number. So the hash function always takes in your key and converts it to a number, OK?

That number, think of it like representing a memory location where you're going to store the value associated with that key. So you're always grabbing a number that represents a memory location. At that memory location, you'll store the value. So next time you want to look up the value associated with a key, you just run that same hash function. The function won't change. You run the same hash function on your object, and you'll be able to get that same integer back. You'll be able to grab that same value back.

But if you're storing mutable objects, that means that object can change. So if you run the hash function, the thing that gives you a number on something that's changed, that number might not be the same because you've changed the thing that you're passing into the function. So why would it give you the same value back?

So let's look at this example. So again, we're storing grades. And let's say we're trying to store a bunch of grades inside our memory. And let's say our memory is just 16 locations, so 0 through 15. So at these locations, I'm going to store grades associated with a person. The function I'm going to run on the student is using their name. So I'm going to store Ana's grade somewhere. But I need to run a function that takes in the string Ana and gets for me a number. That number is where I'm going to store my grade.

So a simple hash function we might do is to say, well, I'm going to take A and map it to 1, B, map it to 2, C map it to 3, and so on. I can sum all of those numbers associated with my letters in my name, 16. And then I can mod it with 16, which is how many entries I have in my memory. So if I mod it with 16, that's going to give me a number 0 through 15, right? If you take the remainder when you divide by 16, you'll either get 0 all the way up through 15.

So if I mod my name, that means I'm going to store my grade at memory location 0. So far, so good? So basically, I made up this hash function that tells me where to put my grade. Now I add another person. Again, I'm going to convert the letters in their name to numbers so that I can easily get a number out of their letters. So I'm basically hashing their letters to a number.

Again, summing this for Eric is 35. I'm going to mod it with 16, which means I'm going to put Eric's grade at location 3. Next person, John, same thing. I add the numbers, mod 16. I'm going to put John at location 15. So this is my memory where I'm storing the values associated with these students.

So if I want to grab back my grade, I run the exact same hash function. So I'm going to run the same hash function on my name. My name hasn't changed, right? It's still the string. I'm not allowed to change it because it's a string. And so I'm going to get the same value back, 0. So that means to grab the letter associated with my name, I just need to go straight into my memory location and look up the value at that memory location. So I know it's going to be a C.

Now, let's say I'm storing a list, a student name as a list. So again, Ana, Eric, and John are immutable, right? They will not change. But if I store Kate as a list, her name might change. Again, I can run the same hash function on her name. That means her grade when I first store it is going to be at location 5. So I'm storing Kate at location 5. All these three strings I know I can get back because they cannot change.

But let's say that Kate goes and changes her name from Kate with a K to Kate with a C. It's the same object, right, the same person. She earned her grade, B, originally. So if I want to grab her grade back, even though her name has changed, I would still like to grab the B associated with her as a person.

But Kate with a C, if I run that same hash function that I ran to put her grade in my table, tells me that I now need to look up her grade at memory location 13, no longer at memory location 5. She's not there. OK, so now it's like did the student disappear and all that stuff.

So you see? Now, that's the reason why I cannot have a mutable object as a key to my list because if that object changes, running that hash function on that changed object might not give me the same memory location where I originally stored the value associated with that object. Does that make sense?

OK, so let's revisit our original example, the one where we tried to store everything in a master list, all these grades in a master list. Now, let's store it in a master dictionary. So I've got my grades. Notice, curly bracket curly bracket is a dictionary. I've got two students in my class, Ana, right? So this is Ana's information and Bob. That's Bob's information. So just two students in my master dictionary.

So the key Ana is going to be one entry key. Key Bob is the other entry key. And what's the information associated with these keys-- well, with Ana, I've got this dictionary associated with her name. So that's this big thing here. I'll explain it in a bit, and similarly with Bob. Bob has one thing associated with him, and it's another dictionary. So I'm mapping strings to dictionaries here. And that's fine because values in a dictionary can be other dictionaries.

So what are these dictionaries about? Well, the number of items in the dictionary for a particular person, there's three elements, right? So comma, comma separates my three elements. The first one is going to be mapped with key mq. The second one mapped with key ps for problem set, and the last one mapped with the string fin for final grade.

So each one of these students has this dictionary associated with them. And that dictionary then itself has three entries, one for the micro quiz, one for the ps, one for the final scores. So now, what's the values associated with those keys? Well, the micro quiz is going to be a list. The problem set is going to be a list. And the final is going to be a string. So a really nice representation of my class, right? And same for Bob.

So now, what if we want to grab a student's exam grade or the student's list of exam grades? Remember that big function with the two nested for loops and the nested ifs? That becomes this line. Isn't that cool? Applause, I like that. Yeah, exactly. We should applaud this because look how easy it is now to-- yes, thank you, yes.

[APPLAUSE]

Dictionaries are awesome, guys. So yeah, so look, that line becomes this-- grabbing one quiz score becomes this single line of code right here. So let's break it down. Again, we do left to right whenever we've got this chain of stuff going on. So the first thing we say is, well, we're looking up grades at some index. So grades at some index gives me that dictionary, so something like this whole thing here, right. OK? Good. That's the first chain.

Now, this box here gets replaced with that dictionary. And I'm doing another index into that dictionary. So that means I'm going to grab the mq associated with that dictionary. So the value associated with mq is going to be this list 544. So this box here gets replaced with the list 544. And then, if I want to grab just the first quiz value, I say now I'm going to index in the list 544 at index 0. So that grabs for me just the 5. So then the first quiz score for Ana was a 5. It's pretty bad.

OK, so let's have you think about this. This is a function, nothing to code here. Just to think. This is a function that grabs the average of every single thing where that thing is what in the class. So if what is mq, as is down here in this example, if what is mq, this code is supposed to get the average of all of the micro quizzes for all the students in the class.

So you basically want to grab the average of 10 plus 8 plus 3. And if it's ps, I would like to grab the average of all the problem sets for all the students in the class. So the average of 10, 10, 7, 8, and 0. So we've got a loop. That goes through every student in the keys. So the student, stud, stud here is going to be this dictionary, right? So given this dictionary, what line should you insert here such that you're creating a list, just a single top-level list of all of the values in there?

So the thing you actually want to end up with, and if we're looking at the ps scores just because it's a little easier to think about, is going to be 10, 10, 7, 8, and 0. So in the end, what I would like to get in my all data, this list that I'm maintaining here, is something like this for the ps.

So think about which one of these lines will accomplish that. And just to help you out we can say student is my dictionary of-- who thinks it's the first one? The second one? Third one? Fourth one? Nobody thought it's the first one? Are you guys sure?

OK, why do you think it's the second one? Is it because of the append? Yeah? Let's think about it. So all data is a list, and what are we appending? So what is data at stud at what going to give us? Data at student is the dictionary, this dictionary here, right, this value here. And if we take this value and index into the what, will it be an integer or will it be a list? A list.

So when we append a list to another list, what is that going to give us? So if we have a list already with a, b, c, if I append another list to this, will it put the element within that list or the list itself?

STUDENT: [INAUDIBLE]

ANA BELL: Yeah, exactly. So that's not going to work for us. Clearly, D is not right either. And definitely, indexing into data at student at what is not going to be right, OK? So that leaves one other choice. The first one. So let's see why the first one works.

We're concatenating, right? So the plus concatenates. So let's say I already have a list, a, b, c. I'm going to concatenate something I already have with data at student at what, which we said is what? Is it a single element or another list?

STUDENT: [INAUDIBLE]

ANA BELL: Exactly, so we concatenate with something like 10, 10, or something like that. So that will return for us a, b, c, 10, 10, which will allow us to do something like getting the sum of all these elements. Questions about that? Does that make sense? Is that all right? Yes.

STUDENT: [INAUDIBLE]

ANA BELL: Yeah.

STUDENT: Why would it [INAUDIBLE]? Like, how come it returns 10 [INAUDIBLE]?

ANA BELL: Because we're indexing into mq. So if you index into mq, mq is your key. So you grab the value associated with that key. So that would be this list here, the 10. Or for ps, it would be the list 10, 10.

So quick recap on lists and dictionaries before we do one final longer example. So again, lists are ordered sequences of elements, right? There is some element at index 0. There's some element at index 1, some element at index 2. So we do have these quote unquote "indices," right? But there's an order to these indices, and there has to be an element index 0 and further up from there, right?

Dictionaries also have these quote unquote "indices," which we call keys. But these are custom. So you can basically rearrange-- you can think of it as being allowed to rearrange indices however you'd like, right? There's no order to the indices in a dictionary. There are some restrictions on the keys or these indices, so they can't be immutable or hashable. But other than that, the things that you store related to that key can be any type, just like you can store any type in a list.

So the last thing I'd like to go through is a larger example. And this will showcase a whole bunch of things that we've been talking about so far. It'll showcase how to, first of all, create dictionaries, which is what we did today. It'll showcase how to reuse functions, how to write functions and reuse functions in other places. It will showcase a little bit of mutability as well. But this is all in a larger example. And if I go a little bit fast through this, I've given you Python tutor links. And it's also in the file to run on your own.

So the goal of this last example is to basically find the most common words in a song's lyrics. And dictionaries are going to be really useful for doing something like this. So I'm going to show you, first of all, what we want to end up with. And then we can talk about how to divide this larger problem into smaller pieces.

So OK, so those are all the pieces. But basically what I want to end up with is I want to have a song be stored as a string, OK? You'll recognize these, but these are very old. I actually haven't updated these songs for a few years. But yeah, anyway, don't judge. So I've got a song stored as a string. And I'm going to run each individual function. But in the end, what I'd like to do is come up with something like this.

So I want to present the user the top-most common words in the song. So here I have a list. So you can see open closed square bracket tells me it's a list. And I've got elements in my list. So here's the first element in my list, which tells the user that the word I occurs 18 times. The next element in my list tells the user that the word we occurs 17 times.

The next element tells the user that the words ain't ever getting older occur all 16 times, and then so on. So we're decreasing in frequency with the most common word occurring 18 times. And then I'm showing the user the most common words down to and including 6. So I would choose some arbitrary value I want to find in the song the words that occur at least six times, for example. OK, so that's the goal of this program.

So how will we achieve this? It's obviously a pretty big task. I wouldn't want to code the entire thing right off the bat. But we can actually divide it into three smaller pieces. The first piece, and we're going to write the code for this, is to create something called a frequency dictionary.

So given a string of words, we're going to create a dictionary that maps each word to how often it occurs. So fancy word, frequency dictionary, but it's pretty simple. It just maps a word to its count inside my long string. So this presents the data which is this string of words in a much nicer format, right? It's a dictionary that tells me the frequency of each word.

Once I have that in hand, things get a little bit easier. I can write another function that finds the word that occurs most often in that dictionary. So the way I'm going to do that is look up the frequencies in the values, find the maximum of those values, and then figure out which keys are associated with that maximum value. And this is all made possible because I've reimagined my data in this frequency dictionary format.

The last step, once I figure out how to write a function that returns for me the words that occur the most times, is to find the words that occur at least some number of times. And I'll go through an example of this one in a few slides when we get to it. But this last function here, number three, you can actually rewrite it in a whole bunch of ways. I'm just going to show you one way to write it that will involve mutability. But you don't have to do it using mutability. You can definitely do it in a whole bunch of-- with a whole bunch of other implementations.

So let's begin by first creating a dictionary that maps the word to their frequencies. So I've picked a song that is a real song. And it has some repetition. And it's short that it fits in one line. So I've got this song here. And I've got my function generate word dictionary. The song is a string.

So it's basically the song a little bit cleaned up, not in terms of words, but in terms of removing punctuation, removing commas, maybe exclamation-- or I might have kept quotations or something like that. But basically, it's removing all of the punctuation and stuff because that will mess up my word counts.

So what is this function going to do given a string for my song? Well, first I'm going to convert all my letters to lowercase. This means that capital T-H-E will be counted as the same word as lowercase t-h-e, which is the correct way to do it. So convert everything to lowercase.

Then I'm going to use our friend, the split function, remember, which takes in my string and splits on a character. So by default, it will split on the space. This puts our string of words in a very manageable format, a list of words. Much nicer to work with lists than work with a string.

Now that I have my word list, I'm going to create my empty dictionary and then populate it. So I'm iterating over my list of words. And then I have a choice. Either I've seen this word already and I want to update the frequency, right? So I want to increase the frequency by 1 because I've already added this word to my dictionary. Or this is the first time I'm seeing this word and I want to add it to my dictionary with a frequency of 1.

So the first case here, the if, will update the frequency because I've already seen the word in my dictionary. So here I'm using in keyword to check if the key, the word, is already in my dictionary. If so, I increase its frequency by 1. Otherwise, this is the first time I'm adding my word to my dictionary. So give it a frequency of 1. And then I return the word dictionary. So this will map strings to integers.

Let's work through it in the Python tutor. So step, step, step, step, step, lowercased my input string, step, I've split it, so now I've got this list of all of my words, step. This is where we begin. So I've created my empty dictionary over here. Keep an eye on this area here. It will become populated soon.

The first word, w, is rah, right? It's the first word in my list. It's obviously the first time I'm seeing it. I have nothing in my dictionary right now. So I'm going to pop in my else, and I'm going to add it to my dictionary with a frequency of 1. Yay, that worked.

Next word in my dictionary, in my list is this one, same word I've already seen. So I'm going to go inside the if and increase the frequency to 2, all right? rah is now 2. Next is ah, right? So here's my word. I've got the next one in my list. It's the first time I'm seeing it. Add it to my dictionary with a frequency of 1.

Next word I'm seeing is, again. Increase its frequency to 2. And I'm going to go faster now. This is increasing the frequency to 3 because I've seen it three times now. And then I'm adding rom for the first time, mah for the first time, and ro for the first time.

And lastly, I'm going to increase mah frequency two more times because it occurs two more times in my song. So it's increased to 2, and now it's increased to 3. And then we're done. So we return the word dictionary. Really nice way to represent my list-- my song, right? Very nice.

OK, so now that I have this frequency dictionary and I've put it up here-- this is what we ended up with-- how can we write a function that returns for me the most frequent word? So one thing we can recognize is the most frequent word has the highest value, Python dictionary value in my dictionary, right? So as a human, I would look to see which one of these entries have the biggest value. As a computer, I can't really do that because I have to do it a little bit more systematically.

So what we can say is, well, let's look at our values and grab the maximum of the values. So here, I'm using this .values function on my dictionary to grab for me all of the values in my dictionary. So this will be the list 2 comma 3 comma 1 comma 3 comma 1. And then I'm running the max function on that list. So max of this list of numbers gives me the maximum value in that list, the 3.

So highest now has the value integer 3. And now, all I need to do is iterate over my entries in my dictionary. So this is kv in the items. And all I'm checking now inside this iteration is if the value is equal to the highest.

So as I'm looking at each entry, is the value for that entry the same as the highest one I've seen? If it is, I'm going to maintain a list of all the words with that highest value because there might be more than one word that has the highest value, as we saw when we actually ran it here, right? Here I had a list of all of these words that occurred 16 times. So that's the output that I want to maintain, OK?

So I'm appending to my words list. And at the end, I'm returning this tuple with the words comma that highest value. So Python tutor, like in the previous time. So let's create our original dictionary. This is what we ended up with last time. So the highest value is 3 here. And I'm going to loop through each entry in my dictionary. So you can say C kv is going to be each one of these in order. So first it's rah w, then it's ah 3, and so on.

Obviously, the 2 is not equal to the 3, so we move on. The 3 equals the 3, so we take the ah, and boom, add it to my list here. So this is the list I'm maintaining of all the words that occur with frequency 3. Next, no for rom, yes for mah, so I'm going to add it to my list. And then no for ro, and I'm done. So the return is going to be this list-- this tuple here with the list of the words that occur three times. OK, good.

Last part, I'm not going to go through Python tutor. I did include a link to it because it becomes very messy with the arrows. But I do encourage you to try to follow it along on your own time. I will explain, however, the way that I chose to solve this problem. So I chose to solve this problem to include mutation and reusing the function that we just wrote that grabs for me the highest value and the words associated with that highest frequency value.

So this is the idea. I have my original word dictionary, right? This is the frequency dictionary we created right off the bat. What I'm going to do is look to see which words occur with the highest frequency. So the highest frequency, my function from before, grabs for me-- it figures out that it's 3. And it figures out the words associated with that 3 are ah and mah. That's exactly what we just did.

So I'm going to grab those words and those entries in the dictionary. And then, I'm going to mutate the dictionary to remove those words because I know those words occur with the highest frequency. So now, I've removed those words, and I've saved them because they were the result of the function that I had just ran, right? So I'm maintaining this frequency list, which will contain all the words that occur at least-- I guess I said greater than one time, so at least two times. So I'm going to grab the ones that occur 3 and 2 times.

So right now, I had just grabbed the words that occur 3 times. I've removed them from my dictionary. So I've actually mutated my dictionary to remove those words. Now, if I run the exact same function that I just wrote on the previous slide on this mutated dictionary, which words will it give me? Which words occur the most now?

STUDENT: [INAUDIBLE]

ANA BELL: Exactly, right? Now the highest value in my dictionary, in this frequency dictionary, is 2 because I mutated to remove what was previously the highest value. So I'm running the same function again on the mutated dictionary to give me just the rah, right?

So I grab that, keep track of that in my frequency list, right, mutate the dictionary to remove that. And as I'm doing that, I'm also keeping track to make sure that the highest frequency I have in the remaining dictionary is at least whatever I was interested in. So here I want it at least 2. So this function, the one I will write, will no longer grab any other values from the dictionary because now frequency 1 I don't want to grab. So this is the resulting value.

And that's the idea. We're using mutability and the function we just wrote to do this task. And this is the code that does that. So this runs the function we wrote previously. Step number 2 gives us that tuple with the list of all the words. This loop here makes sure I still have frequencies that are at least x in the dictionary. I grab the tuple that I just created, so something like this, and add it to my frequency list. So this is the resulting list that I'm keeping track of. And then this bit here removes the word from my dictionary. So I'm mutating the dictionary using this del keyword that we saw at the beginning of this lecture. Yeah, question.

STUDENT: The other one like I know it says whilst basically while the

ANA BELL: Yeah.

STUDENT: I guess what I just [INAUDIBLE] why is still [INAUDIBLE] well the [INAUDIBLE]?

ANA BELL: So I think maybe it's because the function-- I forget what the specification said, but I don't know if it said at least 2 or greater than 2, or at least x or greater than x. It depends on which one I actually said in the specification. But you can imagine changing this to greater than or greater than or equal to. And then we're running this function again inside this while loop to grab the frequency [INAUDIBLE].

Yeah, so these are just the observations I actually stated at the beginning of this example, a bunch of the different things that we've learned that we're using within this example. So slicing or splitting, iterating over the list directly, mutability, using the items, things like that. OK, so that's it. That's all I have. I'll see you guys on Monday. Monday is Halloween. If you'd like to bring a costume, I love Halloween. I will wear something different than what I usually wear.