[SQUEAKING]

[RUSTLING]

[CLICKING]

**ANA BELL:** All right. Welcome, everyone. So in case you missed last lecture, I've got some extra debugging ducks that were left over from last lecture's debugging lecture. Please take them home. I don't want to have to take them back to my office and then bring them back so many times.

So please, give them a good home. If you find them useful in your debugging strategy throughout your programming careers, I suggest you upgrade to carrying a debugging duck with you everywhere, as I have on my phone. I use it in my day to day life. So that's just the next step beyond an actual duck.

All right, so let's get started on today's lecture. We will be going over the idea of exceptions and assertions. And these are basically those scary red errors that we get when our program crashes. OK? Today's lecture will hopefully shed some light on exactly what these exceptions are and how we can actually use them to our advantage in our code.

So let's start by talking about exceptions. So when you run your code, usually it runs without error, produces the right output all the time, like mine does. But sometimes it so happens that your code hits an unexpected condition. And when that unexpected condition is run, you get an exception to something that was expected.

So we've already seen a bunch of different exception examples. So we even talked about a couple of these last lecture. So we've got index errors where you index too far into some list, type errors where you're doing funky things with types that Python doesn't like syntax. Errors are also exceptions. Name errors are also exceptions. So a bunch of these errors that we've encountered are types of exceptions.

So it turns out so far in our programming experience that whenever we get an exception, the program immediately crashes. And really we don't have any way to handle these exceptions. We just accept the fact that it crashed and we go back to the debugging board. But it turns out that in Python we can actually write code to handle these exceptions.

So if your code does happen to throw an exception, so an error occurs or something unexpected happens, you can write code that deals with that situation. And either decides to ignore the fact that an error occurred, set some default values, or just raise your own exception and move on. So we're going to see a bunch of examples of these cases.

So the way that we deal with exceptions is using some code blocks. The way that we handle exceptions is using these try and except blocks. So the way that we write an exception handler is to put some potentially problematic code inside this try block. So the try is a keyword in Python, so obviously you can't name a variable try or anything like that.

If you type it in your code, you'll see it turns blue. And try tells Python that you're starting a code block that contains some lines of code you'd like Python to execute. So just normal code. If Python is able to successfully execute these lines of code without an exception being raised, so without the program crashing, then nothing happens. Nothing is run inside this except block.

And the code just continues as normal. But if it so happens that, in that code that you ran something strange has happened and the code would have crashed. The code actually doesn't crash, because we can catch the exception that gets raised inside this except block. So if we have an associated except block over here to a try block from here.

Python is going to try to run this potentially problematic code. And if an exception is raised, it will stop running any further lines inside the try block and immediately hop to the lines in the except block. And the lines in the except block will then get executed. So to just draw a parallel to, if I were to say this in terms of if and else.

The way that I would describe the try and accept blocks is I would say if, and then I would put all the potentially problematic lines of code that I'd like to write inside this condition for the if. And if all of these lines of code manage to successfully run, then nothing else happens. The inside of this if is essentially just a pass. And we don't execute the else and then we just carry on with the rest of our lives.

But if there is some line of code here that we're trying to run that crashes or that causes the program to crash, Python will say nope, I'm not going to crash just yet, let me see what the code would like me to do. And so we'd hop inside this else and then we'd do something to handle the problem. The something we do is inside this except block.

So again, this is not code we'd ever write. It's just kind of a way to draw parallel with what we know so far. The code that we would write is this try a bunch of code, except, do some lines of code if an error should come up in the try block. So let's look at some examples with code that you should be able to write at this point in the course.

So we have some code on the left here. It's a function called some digits. And we're writing this code without any exceptions OK? We're just writing it as if you were given this code on a quiz. What would be one solution? So this some digits takes in a string S. And we say it's non-empty, containing some digits, and I want to return the sum of all the characters that are digits.

So I don't actually say anything about whether this string S contains non digit characters. But let's write it in a robust way anyway. So we'd have a loop that goes through every character in that string S. And I'm using this in keyword here, this nice little trick here that says if that character, so whatever character it may be, is inside this string of digits, then I know it's a number--

Sorry, I know it's a digit. And I'm going to cast that digits 0 through 9, whatever it may be, to an integer, add it to my running total. And then that loop does its thing until it's done. And then I return the total. So in terms of running the code, this is just as I have it on the slide. So here, if the user gives me the string 1, 2, 3, I'm going to sum 1, plus 2, plus 3, six. Perfect.

And if the user gives me 1, 2, 3, and then some random characters that I know I can't add, Python will still be able to evaluate it, because that if statement will not be run. Right? For A, B, and C, we don't go inside the if. So there's no need to cast anything. All right. So the code still works.

If I didn't have this if here, if I decided to just cast to an int every single thing that comes my way, the first line of code will still work, but the second line of code will throw an exception. You see? I have on the right hand side my scary red text that says ValueError, invalid literal for int with base 10 a. Kind of hard to parse.

But after you've seen a bunch of these, you'll figure out that there's something going on with my types. And then I'm trying to cast-- I'm trying to cast to an integer the string a. Obviously, it doesn't know how to do that. So that's the potentially problematic line. Right? Casting to an integer.

So let's try to write the same code, except that now we'll do it with exception handling. So a bunch of it is going to be the same. What we're going to change is-- The potentially problematic code is these 2 lines here. Right? I don't need the if anymore.

Instead, I'm going to just assume I can cast every single character to an integer. And I'm going to try to do that. So I'm going to try to cast every single character to an integer, and then add it to my running total. Most of the time that's going to work if the input is a digit, but sometimes the users give me something that's non-digit.

In that case, you saw what happens, the code throws a value error. So if we didn't have the except block, nor the try block, the code crashes immediately. No answer is even given. But with the except block, Python will say, oh, in this particular for loop run I had an exception raised. So I'm going to execute whatever is inside the except block.

And the except block says print, can't convert, and then the character that it couldn't convert that time through the loop. And then that loop iteration is done and it goes on to the next character in the sequence. So let's run the code. And this is-- Some digits with the except.

So here I've got the user giving me perfectly fine input. No exceptions are raised. The code worked well. If the user gives me some characters within there, the loops go through. It adds 1, plus 2, plus 3. But then, when it tries to cast to an integer, the a, over here, as the iteration goes to the a, is going to say this raises a value error, as we just saw.

And I'm going to execute whatever is inside the except block. So it prints couldn't convert character. There it is. And then I actually gave the user the character it couldn't convert. It goes on to the next iteration, the next character in the sequence. The B, again, tries to convert B. It can't cast it to an integer. So we print couldn't convert B.

And then lastly, the C, same with the C. Does that make sense? Is that all right so far? So kind of like a little if else situation going on here? Nice. Places to put try/except blocks are when you're dealing with user input, because the users, when they give you some inputs for using the actual input command, they're very unpredictable.

We don't know what kinds of things they'll give you, even though you give them clear instructions to tell me one number or tell me another number. So in these 3 lines of code, down here, I've got the user giving me 2 numbers. And then I print the first one divided by the second one.

So I'm a nice user. I do what I'm told, so I'm going to do 5 divided by 8. Perfect. The code runs well. Let's say somebody else runs the code and this time they decide to do seven divided by, I don't know, five, like that. Value error. So that's one thing that could go wrong. The user tries to be funny.

And then another thing that could go wrong is, let's say, the user gives me a 0 for the second number. So in this case, I get a ZeroDivisionError. You can see Python doesn't know how to divide by 0. So it raises an exception. This thing is ZeroDivisionError.

So this is a case where I'm dealing with potentially problematic inputs. So I'm going to wrap my potentially problematic lines of code in a try and except block. So I've got those 3 lines that I'm going to try to do. And if anything goes wrong, I'm going to execute whatever is in the except block, and all I do is print bug in user input.

OK. So let's run that. That's this one here. So here, again, proper input works well. If the user gives me a letter, bug and user input. So a much nicer friendlier way to crash the program then value error or whatever it was. And then again, if the user gives me a 0, bug in user input. Again, much nicer way to crash the program.

So what we can actually do is have specific behaviors depending on what exceptions are raised. Right? So maybe I don't want a generic text that says bug in user input for both of those cases. Right? Maybe if the user divides by 0, I want to give them a different message than if the user gave me a letter.

So in that case, what I can do is I can have different except blocks for every different error that I might come up. So as I'm writing this code, I can think ahead, right? And I can try to catch specific errors. So here I can catch the value error. So I say except and then I say the name of the error that I'd like to catch.

And this block of code, this except block of code will be run only if the code in the try block crashes with that specific value error. Right? And then I can also catch my zero division error down here. And in this particular case, this except block is only run when the ZeroDivisionError is raised. Right?

So here in the value error I'm going to print for the user, I could not convert to a number. So a more specific error message so they know what's up. And in a ZeroDivisionError I can also flag that there was an issue. I can't divide by 0 by printing that to the screen.

And then I can say a divided by b is infinity. And I can continue the last statement that was supposed to be done in the try block, a plus b, I can give them the answer to a plus b, because we can add a 0 to a. No problem. The last except block down here is catching anything else that's not a value error and not a zero division error.

So I can't think of anything that could go wrong. So if we happen to go in here, something went very wrong. I would say the only thing I can think of is if our computer is almost out of memory, and running this little piece of code pushes it over the edge. Right? Then Python will probably crash and maybe it'll print that error before completely shutting down the computer or something.

But that last one should never really run. So let me show you what happens when we run the code with these specific except blocks now. So if the user gives me perfectly nice input, then the program proceeds as normal. Every line of code inside the try block is executed over here. None of the accept blocks are executed.

The user gives me a letter. I end the program gracefully with the message could not convert to a number so then they know that I caught them. And then the last one is if I try to divide by 0. Again, I've got the little message, can't divide by 0. And then I give them their division to be infinity.

And a plus b is 6. So I do all the lines of code that are caught over here. Questions so far? Seems all right so far. OK. So really nice ways for us to deal with exceptions that get raised in our programs. Now the things that I've told you that we can associate with a try block is an except block, right?

So we've done that. But we can actually associate a couple other things with try blocks. And we don't really use them in this class, but I just thought I'd mention them. So if you have an else block associated with a try block, that means the lines of code inside the else block will execute when everything inside the try block is executed perfectly.

So if everything goes according to plan, whatever you put inside the else block will also get executed. And then you can have a finally block as well. So just like we have a try and accept, you can also have a finally associated with those. And the body of the finally will be executed no matter what.

If an exception was raised, you also execute the finally. If the code worked perfectly fine without raising any errors, the finally also gets executed. So I gave an example here of something that you might put inside the finally block, so sort of clean up code. So if you're writing code that opens files from the file system, a good idea is to close them before you finish your program.

So that's something that you might want to do inside the finally block just close files before shutting down, before your program terminates. OK. So I've shown you how to deal with exceptions, but now what do we do inside the accept blocks? OK.

We've done a couple of different things, mostly printing out that something went wrong, but we can do various other things. One thing, and I don't recommend doing this, is to just fail silently. Certainly we could write code that basically has-- Yeah. There's a question.

**AUDIENCE:** Sorry. The last time when you said--

**ANA BELL:** Yep.

**AUDIENCE:** Like, how is it different from like normal else?

**ANA BELL:** So this is an else that we'd associate with a try. So we would do something like else, and then you would print something here, success or something. And then if the code executes perfectly without an error, then you'll also print success.

**AUDIENCE:** OK.

**ANA BELL:** Yeah. So what do we do inside the except blocks? One thing is to fail silently, which means that well, you could try your entire piece of code, and then you could say except. And then the only line you have in except is maybe a line that says pass.

So that means any error that happens, you would catch, but you do absolutely nothing and let the program continue with a potentially bad value being passed along. That's not a good idea. You could also silently substitute values that you know might be problematic without flagging things happening. Also not good ideas.

At the very least, you should flag something to the output that something weird has happened. Another thing you could do is return some error value. So you could have a whole bunch of variables in your code that you could set to some values, like flags kind of thing, whenever your code runs into an exception block. Right?

But the problem with that is that you have to now check for all these values further on in your code. Right? So now your code becomes overly complicated, because you have a whole bunch of extra variables you're constantly checking to see if any errors were flagged or something like that happened. One last thing, and this is what I'll show you you can do, is to actually still stop the execution.

So much like when we input when we tried to run the sum digits program and it crashed with a value error. We could still make our program crash, but it's on our own terms. So we can raise our own ValueErrors, or whatever kind of error you'd like to raise, with your own custom message. So the code still crashes, which is fine, because maybe you don't want problematic code to move on, but you're basically having it crash with a custom message and a custom error type being raised.

So this is a keyword in Python. You raise your own ValueError and then in parentheses, you put whatever message you would like to put. So here's an example of the sum digits where we raise our own exception. So let's say that indeed, we only wanted the user to give us digits and we don't actually want this function to keep running and passing along the total if the user ever gave us a string that contains letters. Right.

So in that particular case, I'm going to still put a try block and an except block. A try block around the problematic code and except block to catch any errors. But now, instead of printing something and letting the code carry on with the loop, we're going to raise a value error with our own message. So my message is that this drink contained a character.

So if I run this code. And it's actually up here. If I run this code with perfectly fine inputs, there's no issue. Right? We just calculate the total as we want. But if the user gives us some string that does contain extra characters which maybe we don't actually want to have happen, you see, I still have a ValueError, which is the same kind of exception that was raised without the try and except.

But now the message that I've passed in is string contained a character, as opposed to invalid literal for whatever that cryptic message was. So this is a much nicer way to flag or to stop the program, to terminate the program, but do it on your own terms. So let's have you work on this for a couple of minutes.

You'll raise your own ValueErrors. I'd like you to write this function that's called pairwise division. It takes in 2 lists. The lists should be non-empty and they're equal lengths. Right? So per this example, here's 2 lists that are not empty and they're the same length. And I would like the code to basically go element by element.

And create a new list where each element is going to be, for example, 4 divided by 1, 5 divided by 2, and 6 divided by 3. So pairwise, you do the division, put all those elements in a new list, and return that list. If the denominator-- So the second parameter passed in Ldenom contains any 0, raise a value error. OK.

So don't let the code crash with the zero division error, but instead raise a ValueError with a nice message. So start with just the code to do the task, and then add the value error bit at the end. OK. Does anyone have a start how would you like to solve this problem? How do you want to write it? Yeah.

**AUDIENCE:** [INAUDIBLE]

**ANA BELL:** Yep.

**AUDIENCE:** And then try to do the [INAUDIBLE], L [INAUDIBLE] Lnum i over Ldenom i [INAUDIBLE] Sorry, for i, I guess, over the whole [INAUDIBLE].

**ANA BELL:** Lnum at i divided by Ldenom-- whoops-- at i. So we do the division for i, in. So here, what is i? Is it the element or is it an index? Yeah. So how do we grab like basically numbers 0 through the length of one of these lists? If you want to do it. Yeah.

**AUDIENCE:** Oh. So for i in len Lnum.

**ANA BELL:** Yeah. So we have to do range, remember. Yeah, range(len(Lnum)). Yeah. I think those are my parentheses. That's cool that you did list comprehension right away. Does anyone want to rewrite this using a loop? It's true. Oh, yeah. Go.

**AUDIENCE:** For e in Lnum.

**ANA BELL:** Yep.

**AUDIENCE:** Lnum of e-- Or I got to say how L is equal to L.append in [INAUDIBLE].

**ANA BELL:** So we still want to use the index, right? Because if we're looking at the element in Lnum, it's going to be hard for us to grab the same element in Lnum. So let's iterate through 0 through the range. Right? So basically what we did up there, range, len, and then pick one of them because they're the same length.

So now let's change this to i just so we're not confused. I would say i is 0, 1, 2, 3, 4. Right? So now we know this is the index. So with this index in hand, this is the right start. Right? Lnum at i gives me the element in Lnum. Divide by Ldenom exactly at i.

**AUDIENCE:** And then we take that L of [INAUDIBLE]

**ANA BELL:** Yeah. L. We can do L.append, something like that. We can't say L at i equals that, because our L is--

**AUDIENCE:** Not [INAUDIBLE].

**ANA BELL:** --not [INAUDIBLE]. Exactly. Yes. Perfect. So we could do like this. So this is just another way.

And then at the end we can return our variable. Right. OK. So that solves our problem. How do we add the piece where we raise a value error? So how do you want to check that Ldenom has a 0. Because this should hopefully run work with our code without--

Oh. Oops. Did I do it twice? Sorry. Yes I did. Let me just comment one of these out. Oops. There. So how do I add the piece about valuers. Yes.

**AUDIENCE:** Depend on your try. And then you add except. And then [INAUDIBLE] do not contain zero.

**ANA BELL:** Yep. So we can pop this into a try and then except and raise ValueError. Yep. And with some nice message here. Nice message. And we can also put the entire for loop under the try. The code is not very long. It wouldn't make a difference.

So if we try to run it like that now, I've got my value error with my nice message. Yeah. Another way we could raise the ValueError, just for completion's sake, is to say something like-- You can even raise things inside if statements. They don't have to be part of except blocks. Before we even do anything with the code, we can say if 0 is in Ldenom, raise value error.

That would also be a fine thing to do. All right. So we can raise valuers wherever we'd like. So now I'd like to talk a little bit about assertions. So assertions are actually still exceptions that get raised. They're just a very special type of exceptions that we mostly use as a defensive programming tool.

So assertions are basically used to enforce these contracts that we make between somebody who writes a function and somebody who uses a function. So that's basically the function docstrings. Right? When we talked about docstrings, I said that it's very hard for us to enforce the text within the docstring. Right?

Because the person who's writing the function is saying the input list should not be empty or to input lists, like in the previous example, should be the same length. And there's no way for us to really enforce that. It's just something that's nice to have. And we're going to guarantee that the code runs if these things are upheld.

But it turns out that assertions are actually a nice way for us to add to a nice thing to add to our functions that do try to enforce this contract through the specification. So the way we create an assert, we say, assert and I'm asserting that this statement is true. So if I want that the input length for a function to be non-zero, I would assert that the length L is not equal to 0 or something like that.

And if the assertion is true, if that condition is met, then the code carries on as normal. But if the assertion is not true, then Python ends with an assertion error and then some message that the condition was not true. And these are really nice because it halts the execution of a program as soon as that contract is not held. Right? As soon as something within the specification has gone wrong, then the program terminates with those assertion errors.

And it's nice to see them, because if you're debugging your code, you don't want to propagate bad values through functions, because that value might get propagated later and later, and later, and then it would make your debugging very hard. OK? So if you stop the execution as soon as something is just strange or off, as in something like an assumption a parameter is not met, then that's good.

So in some digits example, here is the code that we wrote last. So total down to the bottom is exactly what we had before. All we're going to add is this assert statement up here that the length of is not empty, because part of my contract here is that s is a non-empty string. Right? So that's a nice thing to assert.

If the user ever gives us an empty string, the program will terminate. So in this example here, I've got some digits with the assert. So if the user gives us an empty string, no total was created and the assert was immediately false. So length was equal to 0. The assertion error was raised with the message s is empty. So what I had here.

If I have fine input, then no assertion is raised, and the code carries on as normal. So that's nice. So let's have you add one more line of code to the program that we just wrote. Just add an assert statement that enforces the contract. So I have Lnum and Ldenom are non-empty lists of equal lengths.

So you can do this all in one assert statement, or you can put 2 separate assert statements with 2 separate messages. However you'd like, it is fine with me. So I'll give you a minute to work on that and then we can write it. All right. What assertions should I put in here? Yeah.

**AUDIENCE:** Maybe assert len(Lnum) and then [INAUDIBLE] Ldenom, and then [INAUDIBLE].

**ANA BELL:** So the thing I'm asserting should be true. So do I want them to be equal? Yes. Exactly. So I want len(Lnum) to equal len(Ldenom) Yeah. Ldenom, like that. Yep, that's one. Right? So the thing you want, you're asserting that this is true, and if not, comma, we're going to put a message. Right?

Lengths different or something like that. Do you want to do the other assert statement? Or does somebody else want to take a crack at the other assert? So the other one is that they are non-empty lists. Yeah.

**AUDIENCE:** And assert at the length of Lnum is not like a exclamation point. Same thing for Ldenom. And that should take out [INAUDIBLE].

**ANA BELL:** Yep. So we can definitely do that. Not equal to 0, comma. Empty list or something like that. Yep. Very nice. So here we're trying to enforce our nice contracts. And I've got 2 examples down here.

So here I've got 2 different lengths of lists. So there you go. My assertion was raised with the nice message, lengths differ. And then the code would immediately stop and it would force us to check to see why these links are different. So these bad lists won't propagate any further if I had larger pieces of code.

And then same here. I've got this assertion error that I have an empty list. Any questions so far? OK. One more example I want to go through. I'm not going to actually run this one, but it is in the Python slides. I just wanted to give you another example of how we can use exceptions and assertions in just a different setting.

And it hopefully shows that as a programmer, you get to choose how you add these exceptions and assertions. Right? So wherever they seem reasonable to add, you should add them. So in this particular example, we are assuming that we have a class list. In this case, I only have 2 students in my class.

So these are their test grades. So I've got a list that looks like this. It looks complicated, but I'll walk you through it. This is one student in my list. And this is another student in my list. So I've got a list of lists, where these things that I've highlighted in red is my students. And for each student I have more lists as part of their information.

So the first list related to one student is their name. Right? The first element is their first name. Second element is their last name. And then the second list for that student is their grades in the class. So just another list of all the grades in the class. OK. So what I would like to do, and this is the code I'm going to go through, is what is I'd like to create a new list based on the original GRE test grades that contains the same information as before.

So you can see, I still have 2 lists of students, the first row and the second row. And in each student's information I again still have their name and their list of grades, but I'm adding one more item at the end for each student, which is the average of the list of grades. So I've taken the average of these and plopped it as my integer or float at the end.

And same with my next student. So the code that's going to do this looks like this. That's just the original list to give you an example to look at, because I find it hard to see things without examples. So this is the code that gets the stats for the class. So that creates this new list containing my average at the end for each student in the class list.

So for example, student here, stu is going to be this list of 2 lists. What I'm going to do in my new list that I'm creating here, new stats, is I'm going to append student at 0, which is just their name. So just a straight copy and paste. Student (1), which is just their grades. Again, a straight up copy and paste of all their list of grades.

And then I'm going to apply a function named average to the list of student grades. And what we're going to see in the next couple of slides is we're going to see a few variations of this average function and what happens when we apply these different functions. But for now, I will assume that this code will do the job. So the original average function will take in a list of grades.

So this grades here will look like this blue box here. Right? So just the list of numbers. It's going to sum all the grades, so sum of all the elements inside that list and divide by how many there are average. OK. Now let's say that I have a student that contains no quiz grades or no grades at all.

In that case, their list of grades will be empty. So if I try to apply the sum of all the grades, divided by the length of the grades, for somebody who has no grades information, that length will be 0. So I'm going to get a ZeroDivisionError when I run my code, and it will crash.

So what I'd like to do is to change my average function to try to catch such an error. So I'm going to try to do the sum divided by the length. And I'm going to catch this ZeroDivisionError inside this except block. And all I'm going to do is print warning: no grades data. So for any student that actually has grades information here, the code will work-- the code to get the average will work just fine, because it will do the sum divided by the length.

So that means the try block will succeed, and we're going to return the sum divided by the length. But if any student enters the zero division error here, we're going to print something. And what do we return? What is the function return if we enter the except block?

That's what's going to be printed, but what is this function actually return? What value? None. Exactly. Right? There's no return inside this except block. And no code after it either. So you can see here, if it successfully completes for these 3 students, we've got our numbers. That's what we returned. But for the last student that has no grades information, we're returning none.

OK. I don't like that. What I would like in my grades book is to have numbers as my value there. So instead, let's add a return for that except block. So we're still going to flag the error. We still want to know that something weird has happened. I don't just want to return a 0 without actually telling the user that something's gone down.

I still flag the error, but then I can return a 0, so that it's still a number. And then, if I do something with numbers at the end, then it all works out. This was a particularly hard first Quiz, 10, 10, 80.

OK. One last thing we can do is to just assert. Right? So if we want to make sure that every student had some sort of grades information, maybe if the grades data was empty, something weird happened from a previous function that might have been called, I don't know, but maybe we say let's just assert that the length of the grades is not 0, so we only want this code to execute if there are some grades information.

And if not, let's just raise an assertion error just in case. So we can assert that the length of grades is not equal to 0. And in that case, the code will terminate as soon as we try to get that last student information. It will crash and it will crash with this assert statement that there's no grades data.

And then we can go back to the code and see, did we actually expect the student to have information or not? And then we can try to work through. So just a quick summary of exceptions and assertions. Hopefully, this lecture kind of demystified some of these exceptions that we've been getting.

It showed you they're not as scary, as they might have seemed originally. But they don't always have to terminate the program. Right? You can catch them. You can deal with them in whatever way that makes sense for that particular function or program. You can print a nice output to the screen.

You can set some default values. You can still terminate the program, but do it on your own terms with your own errors, with your own custom messages, so that the users can see something nicer than the default Python messages. Right? And so exception handling is a very important part of writing a program, especially if you expect weird things to happen. Right?

Assertions, on the other hand, are a type of exception. And they're useful, as I've mentioned, to try to enforce these contracts, these specifications. You basically use assertions because you don't want bad values to propagate. So as soon as something that isn't as you expect it to be happens, assertion error is raised and the program immediately terminates, allowing you to check to see why exactly those conditions were not met.

OK. So that's it. That's all I had.