PROFESSOR: All right, everyone. Good afternoon. Let's get started. So today's lecture will be on testing, debugging, and then exceptions and assertions. So before we begin, let's start with an analogy to sort of come back to real life for a second.

So I've made soup before. Perhaps you've made soup before. Let's say you're making soup in this big pot here. And it turns out that bugs keep falling into your soup from the ceiling. All right. Quick question to the audience. What do you do if you encountered this issue?

AUDIENCE: [INTERPOSING VOICES]

PROFESSOR: All right. Hands up. One one at a time. Anyone have any idea? Yeah.

AUDIENCE: Eat it.

PROFESSOR: Eat it. You want to eat it Anyway OK. All right. We're going for an analogy here with computer programming. I don't know what you'd do if you have a buggy program, I guess you just release it to the customer and they'd complain, but. OK. What else? Yeah.

AUDIENCE: [INAUDIBLE] Cover the soup?

PROFESSOR: Cover the soup. That's a good suggestion. Yeah. So you can cover the soup, so put a lid on it. Sometimes you'd have to open up, take the lid off, right, to check to make sure it's done. To taste it, add things. So bugs might fall in in between there. But covering the soup is a good idea. What else. Yeah.

AUDIENCE: Debug it.

PROFESSOR: Debug it. I wish I had something for that answer. All right. That's a good answer. Yeah.

AUDIENCE: Take all the food out of your house so there's no-- nothing for the bugs to eat.

PROFESSOR: So take all the food out of your house so there's nothing for the bugs to eat. That's sort of the

equivalent of cleaning, like doing a mass cleaning of your entire house. That's a good, that's a good one. That's sort of eliminating the source of the bugs, right? What else? Yeah, John.

AUDIENCE: Decide it's high protein and declare it a feature.

PROFESSOR: Decide it's high protein and declare it a feature. That's probably what a lot of people would do, right? All right. Cool. So I wish computer debugging was as fun as taking bugs out of your soup. So what did we decide? Well we could check the soup for bugs. Keep the lid closed, that was a good suggestion. And cleaning your kitchen, which someone suggested. The equivalent of cleaning their kitchen was to just throw out all the food. I would take a mop and clean the floor, but yeah, that works too. So we can draw some parallels for this analogy with computer programming.

So checking the soup is really equivalent to testing, right? You have a soup you think has bugs in it. Test it. Make sure there's no bugs. Continue on. Keeping the lid closed. It's sort of this idea of defensive programming. So make sure that bugs don't fall in in the first place. Sometimes you have to open the lid to make sure that the soup is tastes good or whatever. So that's equivalent to defensive programming. So try not to have bugs in the first place, but they might show up anyway. Cleaning the kitchen is eliminating the source of the bugs in the first place. This is actually really hard to do in programming. But you can still try to do it. OK.

So let's talk a little bit about programming so far in 60001 600. So you expect, really, that you write a program, you maybe do a little debugging, and you run the program and it's perfect. Right? You just nailed it. But in reality you write this really complex piece of code and you go to run it and it crashes. Right? It's happened to me many times. It's happened to you many times. That's the reality. OK. So today's lecture will go over some tips and tricks and debugging and how you can help make your life easier when you're writing programs so you don't end up like this little girl here. Disappointed beyond belief. All right.

So at the heart of it all is really starting with a defensive programming attitude. OK. And this comes back to the idea of decomposition and abstraction that we talked about when we started-- when we did the lecture on functions. Right? So try to start out with two modularize your code, right? If you write your code in different blocks, documenting each different block, you're more likely to understand what's happening in your code later on and you'll be able to test it and debug it a lot easier.

Speaking of testing and debugging, once you've written a program that's modular, you still

have to test it. And the process of testing is really just coming up with inputs. Figuring out what outputs you expect. And then running your program. Does the output that the program give match what you expected? If it does, great, you're done. But if it doesn't, you have to go to this debugging step. And the debugging step is the hardest part. And it's really just figuring out why the program crashed, or why the program didn't give you the answer that you expected it to give.

So as I mentioned, the most important thing is to do defensive programming and to that end, you want to set yourself up for easy testing and debugging. Which really comes down to making sure that the code you write is modular. So write as many functions as you can. Document what the functions do. Document their constraints. And it'll make your life a little bit easier later on when you have to debug it.

When do you want to start testing? Well first you have to make sure your program runs. So eliminate syntax errors and static semantic errors which, by the way, Python can easily catch for you. Once you've ensured that a piece of code runs, then you want to come up with some test cases. So this is pairs of input and output for what you expect the program to do.

Once you have your test cases and a piece of code that runs, you can start doing tests. So there's three general classes of tests that you can do. The first is called unit testing. And if you've written functions, unit testings-- testing just makes sure that, for example, each function runs according to the specifications. So you do this multiple times. As you're testing each function, you might find a bug. At that point, you do regression testing. Come up with a test case that found that bug. And run all of the different pieces of your code again to make sure that when you fix the bug, you don't re-introduce new bugs into pieces of the code that had already run.

So you do this a bunch of times. You do a little bit of unit testing, a little bit of regression testing, and keep doing that. At some point, you're ready to do integration testing. Which means, test your program as a whole. Does the overall program work? So this is the part where you take all of the individual pieces, put them together. And integration testing tests to make sure that the interactions between all of the different pieces works as expected. If it does, great, you're done. But if it doesn't, then you'll have to go back to unit testing, and regression testing, and so on. So it's really a cycle of testing.

So what are some testing approaches? The first, and this is probably most common with

programs that involve numbers, is figuring out some natural boundaries for the numbers-- for the program, sorry. So for example, if I have a function is_bigger, and it compares if x is bigger than y, then some natural boundary, given the specification, is if x is less than y, x is greater than y, x is equal to y. Maybe throw in less than or equal to or greater or equal to, and so on.

So that's just sort of an intuition about the problem. It's possible you have some problems for which there are no natural partitions. In which case, you might do some random testing, and then the more random testing you do, the greater the likelihood that your program is correct. But there's actually two more rigorous ways to do testing. And one is black box testing and the other one is glass box testing.

In black box testing, you're assuming you have the specifications to a function. So that's the docstring. All you're looking at is the docstring and coming up with some test cases based on that. In glass box testing, you have the code itself and you're trying to come up with some test cases that hit upon all of the possible paths through the code.

All right. Let's look at an example for black box testing. I'm finding the square root of x to some close enough value given by this epsilon. And the idea here, notice I don't actually give you how this function's implemented. The idea is that you're just figuring out test cases based on the specification. And the great thing about black box testing is that whoever implements this function can implement it in whatever way they wish, they can use approximation method, that can use bisection method, it doesn't matter. The test cases that you come up with for this function are going to be exactly the same. Right? No matter what the implementation.

So for this particular function, here's a sample set. We check the boundary, we check perfect squares, we can check some number that's less than 1, we can check maybe irrationals, and then you do extreme tests. So when either epsilon is really large or epsilon is really small, or x is really large or x is really small, and all the possible combinations of those. So the important thing about black box testing is that you are doing you are creating the test cases based on the specifications only.

Glass box testing, you're using the code itself to guide your test cases. So if you have a piece of code and you come up with a test case that goes through every single possible combination of input-- of every single possible path through the code, then that test set is called path complete. The problem with this is when you encounter loops, for example. Every single possible path through a loop is maybe the code not going through the loop at all, going

through once, going through twice, going through three times, four times, five times, and so on. Right? Which could be a very, very big test.

So instead there are actually some guidelines for when you're dealing with loops and things like that. So for branches, when you're doing glass box testing, it's important-- you should just exercise all of the parts of the conditional. So make sure you have a test case that goes through each part of the conditional. For for loops, make sure you have a test case where the loop is not entered at all, where the loop is entered one time, and when the loop is entered just some number more than once.

For while loops, similar to for loops, except that make sure you have test cases that cover all of the possible ways to break out of the while loop. So if the while loop condition becomes false, or if maybe there's a break inside the while loop, and so on.

So in this example, we have the absolute value of x. This is its specification and this is the implementation that someone decided to do for this function. So a path complete test set means that you want to have a test that goes through each one of these branches. So if x is less than minus 1, well, minus 2 is less than minus 1. So that's good. And otherwise, which means pick a number greater than minus 1. So 2. So 2 and minus 2 are path complete. Yield path complete-- yields a path complete test suite.

But notice that while we've hit upon every possible path through this code, we've actually missed a test case. Minus 1. So this code incorrectly classifies minus 1 as returning minus 1, which is wrong. So for glass box testing, in addition to making sure you're going through all the possible paths through the code, you also want to make sure you hit upon any boundary condition. So in this case, for branches, minus 1 is a boundary condition.

So you've created a test suite, you've tested your program, chances are you found a bug. What do you do now?

All right. Quick sort of detour into a little bit of history. The history of debugging. So 1947, this computer was built. And it was a computer that was very impressive for its day. It could do things like addition in 0.1 seconds. Things like multiplication in 0.7 seconds. And take the log of something in five seconds. So faster than a human, possibly. But pretty slow for today's standards. And a group of engineers were working on running a program that found-- that was supposed to find the trigonometric function. And among them being this-- one of the first female scientists, Grace Hopper.

And they found that their program was not working correctly. So they went through all of the panels and all of the relays in the computer, and they isolated a program in panel F relay 70, where they found this moth. Just sitting in there. I think it was dead, probably electrocuted. But it was a moth that was impeding the calculation. And I don't know if you can read this, but this part right here. They made a note in their logbook that says, first actual case of bug being found. Which I think is really cute. So they were literally doing debugging in this computer. Right. All right.

So you won't be doing that sort of debugging. You'll be doing a virtual kind of debugging in your programs. Which, again, is not that fun. But you still have to do it. So debugging, as you might have noticed so far in your problem sets, has a bit of a steep learning curve. And obviously your goal is to have a bug free program, and in order to achieve that, you have to do the debugging.

There are some tools which some of you have been using. There are some tools built into Anaconda, or whatever ID you've been using to do debugging. I know some of you have been using the Python tutor, which is awesome. The print statement can also be a good debugging tool. But over above everything else, it's really important to just be systematic as you're trying to debug your program.

I want to talk a little bit about print statements and how you can use them to debug, because I think-- Python tutor, if you don't have the internet, you might not be able to use it. If you don't know how to use the debugger, you don't need to learn. But print statements, you'll always have them, and you can always put them in your program. And they're really good ways to test hypotheses.

So good places to put print statements are inside functions. Inside loops, for example, what are the loop parameters, what are the loop values, what function-- what functions return what values. So you can make sure that values are being passed-- the correct values are being passed between parts of your code. I will mention that you can use the bisection method when you're debugging. Which is interesting.

So if you take a print statement, find approximately the halfway point in your code. Print out what values you-- print out some relevant values. All of the possible-- print out some values at that point in your code. If everything is as you expect it to be at that point in your code, then you're good. That means the code so far is bug free. That means that-- however, that means

that the code beyond it has a bug, right?

So since you've put a print statement halfway in your code and you think that gave good results, then put a print statement 3/4 of the way in the code. And see if the values are as you expect at that point. And if they are, great. Then put a print statement further down. So in this way you could use the bisection method to pinpoint a line, or a set of lines, or maybe a function that that's giving you the bad results.

So the general debugging steps is to study the program code. Don't ask what is wrong, because that's actually part of the testing. So your test cases would have figured out what's wrong. The debugging process is figuring out how the result took place. And since programming is-- programming and debugging is, sort of, is a science, you can use the scientific method as well. So look at all the data, that's your test cases. Figure out a hypothesis. Maybe say, oh, maybe I'm indexing from 1 instead of 0 in lists, for example. Come up with an experiment that you can repeat. And then pick a simple test case then you can test your hypothesis with.

So as you're debugging, you will encounter error messages. And these error messages are actually pretty easy to figure out. And they're really easy to fix in your code. So for example, accessing things beyond the limits of the lists give you index errors. Trying to convert, in this case, a list to an integer gives you type errors. Accessing variables that you haven't created before gives you name errors. And so on and so on. And syntax errors are things, for things like, if you forget a parentheses, or forget a colon, or something like that.

So error messages are really easy to spot. The Python interpreter spits these out for you and then you can pinpoint the exact line. Logic errors are actually the hard part. And logic errors are the ones that you will be spending the most time on. For which I would recommend always trying to take a break. Take a nap, go eat. Something. Sometimes you'd have to start all over, so throughout the code you have and just sit down with a piece of paper, try to figure out how you want to solve the problem.

And if you look up the term rubber ducky-- a lot of heads went up on that one-- rubber ducky debugging. That is an actual term in Wikipedia. And it's when a programmer explains their code to a rubber ducky. That's me on the left explaining code to my rubber ducky. You should always-- you should go buy one. Or code to anyone else, preferably someone who doesn't really understand anything. Because that'll force you to explain everything really, really closely.

And as you're doing that, you'll figure out your problem. And I figured out my problem in both of these cases. So just go back to the basics.

Quick summary of dos and don'ts of debugging and testing. So don't write the entire program, test the entire program, and debug the entire program. I know this is really tempting to do, and I do it all the time. But don't do it. Because you're going to introduce a lot of bugs and it's going to be hard to isolate which bugs are affecting other ones. And it'll lead to a lot more stress than you need. Instead do unit testing. So write one function, test the function, debug the function, make sure it works, write the other function, and so on and so on. Do a little regression testing, a little more unit testing, a little integration testing, and it's a lot more systematic way to write the program. And it'll cut down on your debugging time immensely.

If you're changing your code, and inevitably you'll be changing your code as you're doing your problem sets, remember to back up your code. So if you have a version that almost works, don't just modify that and maybe save a copy. [INAUDIBLE] you've got terabytes of memory on your computer, it won't hurt to just make a quick copy of it. Document maybe what worked and what didn't in that copy. And then make another copy, and then you can modify your code.

So that's sort of a high level introduction to testing and debugging. The rest of the class will be on the error messages, or on errors that you will get in your programs. So when your functions-- when you run functions, or when you run your program, at some point, the program execution is going to stop. Maybe it encountered an error because of some unexpected condition. And when that happens you get an exception. So the error is called an exception. And it's called an exception because it was an exception to what was expected. To what the program expected.

So all of these errors that I've talked about in the previous slides are actually examples of exceptions. And there are actually many other types of exceptions, which you'll see as you go on in this course and also in 60002.

So how do we deal with these exceptions? In Python, you can actually have handlers for exceptions. So if you know that a piece of code might give you an error. For example, here I'm dealing with inputs from users. And users are really unpredictable. You tell them to give you a number, they might give you their name. Nothing you can do about that. Or is there? Yes there is. So in your program you can actually put any lines of code that you think might be

problematic, that might give you an error an exception, in this try block. So you say try colon, and you put in any lines of code that you think might give you an error.

If none of these lines of code actually produce an error, then great. Python doesn't do anything else. It treats them as just part-- as just if they were part of a regular program. But if an error does come up-- for example, if someone doesn't put in a number but puts their name in-- that's going to raise an error, specifically a value error. And at that point, Python's going to say, is there an accept statement? And if so, this except statement is going to handle the error.

And it's going say, OK, an error came up, but I know how to handle it. I'm going to print out this message to the user. So if we look at code-- this is the same code as in the slides-- and there's no try except block around it. So if I run it and I say, three and four, it's going to run fine. But if I run it and I say, [INAUDIBLE] a, it's going to give a value error.

Now if I run the same piece of code with try-- with a try except block. I run it, if I give it regular numbers, it's fine. But if I'm being a cheeky user, and I say three, automatically this would have raised the value error in the previous version of the program. But in this version of the program, the programmer handled the exception or caught the exception, and printed out this nicer looking message. So bug in user input is nicer than this whole lot here. A lot easier to read.

So any problematic lines of code, you can put in a try block, and then handle whatever errors might come up in this except block. This except block is going to catch any error that comes up. And you can actually get a little bit more specific and catch specific types of errors. In this case, I'm saying, if a value error comes up-- for example, if the user inputs a string instead of an integer-- do this, which is going to print this message. If the user inputs a number for B such that we're doing a divided by b, so that would give a 0 division error. In that case we're going to catch this other error here, the 0 division error, and we're going to print this other message, can't divide by 0.

So each-- so you can think of these different except blocks as sort of if else if statements, except for exceptions. So we're going to try this. But if there's a value error do this. Otherwise, if there's a division error, do this. And otherwise do this. So this last except is actually going to be for any other error that comes up. So if it's not a value error, nor a division error, then we're going to print, something went very wrong. I couldn't even try to create-- I couldn't even try to

make the program come up with any other error besides those two.

So a lot of the time you're just going to use try except blocks. But there's other blocks that you can add to exceptions. And these are more rarely used, but I'll talk about them anyway. So you could have an else block. And an else block is going to get executed when the code in the try block finished without raising an error. And you can also have a finally block, which is always executed. If the code in the try block finished without an error, if you raised an exception, if you raised a different kind of exception, if you went through the else, in any of these cases, whatever's in the finally block is always going to get executed. And it's usually used to clean up code. Like if you want to print, oh, the program finished, or if you want to close a file, or something like that.

So. We've encountered errors. We've caught them. What else can we do with errors-- with exceptions. Three other things. So one is if we've caught an error, we can just fail silently. What this means is, you've caught an error, and you just substitute whatever erroneous value the user gave you for some other value. That's not actually a very good idea. That's a bad idea. Because suddenly the user thinks that they entered something, and they think everything's great, your program accepts it, but then they get some weird value as an output, which is far from what they expected. So it's not really a good idea to just replace user's values with anything.

In the context-- so this is in the context of a function. In the context of a function, what else can we do? Well, if you have a function that fails, for example, let's say you're trying to do you're trying to get the square root of an even number. And let's say the user gives you a-- sorry, you're trying to find the square root of a positive number. And let's say the user gives you a negative number. Well, if the user gives you a negative number, your function could return an error value, which means, well if the number inputted is less than 0, then return 0. Or minus 1. Or minus 100. Just pick any value to return which represents some error value.

This is actually not a good idea either, because later on in your program, if you're using this function, now you have to do a check. And the check is, well if the return from this function is minus 1 or minus 100, do this. Otherwise, do this. So you you're complicating your code because now you always have to have this check for this error value. Which makes the code really messy.

The other thing we can do is we can signal an error condition. So this is how you create

control flow in your programs with exceptions. So in Python, signaling an error condition means raising your own exception. So so far we've just seen the programs crashing, which means they raise an exception and then you deal with them. But in this last case, you're raising your own exception. As a way to use that exception later on in the code. So in Python, you raise your own exception using this raise keyword and then an exception. And then some sort of description, like "user entered a negative number" or something like that.

A lot of the time we're going to raise a value error. So if the number is less than 0, then raise a value error, something is wrong. The key word, the name of the error, and then some sort of descriptive string.

So let's see an example of how we raise an exception. I have this function here called get ratios. It takes in two lists, L1 and L2. And it's going to create a new list that's going to contain the ratio of each element in L1 divided by each element in L2. So I have a for loop here. For index in range length L1. So I'm going through every single element in L1. I'm going to try here. I'm going to try to do this line. So I think that this line might give me an error. So I'm going to put it in a try block. The error I think I'm going to get is a 0 division error, because what happens when an element and L2 is 0?

And when an element in L2 is 0 I'm going to append this not a number as a float. So NAN, as a string, you can convert it to a float, and it stands for not a number. So then I can continue populating the list with these not a numbers. If an element and L2 is 0. And otherwise, if there's no 0 division error, but there's another kind of error, then I'm going to raise my own error. And say, for any other kind of error, just raise a value error. Which says, "get ratios was called with a bad argument." So here I'm sort of consolidating all errors into my one value error. So later on in my program, I can catch this value error and do something with it.

Here's another example of exceptions. So let's say we're were given a class list. We have a list of lists. Where we have the name of a student, first name and last name, and their grades in the class. So we currently have two students. And what I want to do is create a new list which is the same things, the same inputs here. But I'm adding an extra-- I'm appending an extra value at the end of the list for each student, which is the average of all of their grades. Or all of their-- yeah, all of their grades.

So let's look at the code. This is the function that takes the class list, which is this whole list here. I'm creating a new list inside it, initially empty. And then I'm going for every element in

the class list. I'm appending element at 0, which is going to be this first list here. So it's going to be the first name and the last name. Element at 1, which is the grades. And then the last thing I'm appending is a function call. The function call being called with element 1, which is all of the grades, and this is my function call. We're going to see three different function calls.

This is the first one. It simply takes the sum of the grades and divides it by the length of the grades. If these students are responsible, and they've taken all of the tests, then there's no problem. Because length of grades is going to be something greater than 0. But what if we have a student in the class who didn't show up for any tests? Then we have no record of any of their tests. No record of grades or anything like that. So they're going to have an empty list. So if we run this function, averages, on their data, we're actually going to get a 0 division error, because we're trying to divide by length of grades, which is going to be 0.

So what can we do? Two things, two options here. One is we can just flag the error and print the message. So here there's a new average function, an improved one, that's going to try to do the exact same line as the previous one. And it's going to catch the 0 division error. And when it catches it, it's going to print this warning. And when we run it, we're going to get, "warning, no grades data," which is fine. And we're going to get this "none" here, for the grades. So everyone else's grades was calculated correctly, and for this last one, we got a none.

That's because, when we entered this except statement, if this is a function, remember functions return something. This function in this particular except statement didn't return anything. So it returns a none. So for the averages for this particular function, the average is going to be a "none" for this person who didn't have any grades associated with them. And yeah, so that's basically what I said. So that's our first option, is to just flag the error and print a message.

The other option is to actually change the policy. So this is where you replace the data with some sort of default value. And if you do something like this, then this should be documented inside the function. So when you write the docstring for the function, you would say if the list is empty, then it'll will a 0. So this is the exact same thing as before. We have a try and an except for the 0 division error. We also print a warning, no grades data. And then we return the 0. So we still flag the error, and now instead of a "none," we get a 0, because we've returned 0.0 here, as opposed to just leaving it blank.

All right. So those are exceptions. Last thing we're going to talk about today are these things called assertions. And assertions are good example of defensive programming. In that, you have assert statements at the beginning of functions, typically. Or at the end of functions. And assert statements are used to make sure that the assumptions on computations are exactly what the function expects them to be. So if we have a function that says it's supposed to take in an integer greater than 0, then the assert statement will assert that the function takes in an integer that's greater than 0.

Here's an example. This is the same average function we've seen before. Here, instead of using exceptions, we're going to use an assert statement. And the assert statement we're putting right at the front. At the beginning of the function, sorry. And the key word is assert. The next part of the assert is what the function expects. So we expect that the length of grades is not equal to 0. So has to be greater than 0. And then we have a string here, which represents what do you print out if the assertion does not hold.

So if you run the function, and you give it a list that is empty, this becomes false, so the assert is false, and we're going to print out an assertion error, no grades data. If the assert is false, the function does not continue. It stops right there. Why does it behave this way? Well, assertions are great to make sure that preconditions and post-conditions on functions are exactly as you expect. So as soon as an assert becomes false, the function's going to immediately terminate. This is useful because it'll prevent the program from propagating bad values.

So as soon as a precondition isn't true, for example, as you enter a function, then that means something went wrong in your program. And the program is going to stop right there. So instead of propagating a bad value throughout the program, and then you getting an output that you didn't expect, and then you having to trace back to the function that gave this bad value, you'll get this bad value, you'll get this assert being false a lot earlier. So it'll be a lot easier to figure out where the bug came from. And you won't have to trace back so many steps.

So this is basically what I said, you really want to spot the bugs as soon as they're introduced. And exceptions are good if you want to raise them when the user supplies bad data input, but assertions are used to make sure that the types and other-- the types of inputs to functions, maybe other conditions on inputs to functions, are being held as the values are being passed in. So the keyword here is making sure that the invariants on data structures are meant. And

that's it. Great. Thanks.