PROFESSOR: For those of you who are unaccustomed to seeing it, that blue stuff through the window is called sky.

OK. I left you last time with a question. I'd shown you a bisection search implementation, or I should say putative implementation of the square root, and told you that it was flawed, and asked you to think about what was wrong with it. So first, I'd just like to take a poll.

How many of you know what the problem is, or at least what a problem is? OK well that's a good start. And I won't ask how many of you don't know, because I presume that's the rest of you. So let's look at it now. I'm not going to ask you the answer, because I want to show people how to find it.

So here's a slightly simplified version of it. Just get rid of this stuff here, so it doesn't confuse the picture. So this is roughly the one we looked at on Tuesday, but I just took out some print statements and things and simplified it. And let's just run it. And we'll run it with trying to find the square root of 0.5. See what happens when we run it. Well, not much happens when we run it. Actually quite a lot is happening. We just can't see it.

So this program is running longer than I expected it to. So if you go to the keyboard and you hit Control and C. It will interrupt the program. It generates what's called a keyboard interrupt, and it stops it. And it tells us where it happened to stop it. It happened to stop it in line seven-- the test of the while loop.

And the problem is, the program just seemed to be running forever. So despite my, perhaps persuasive, argument last time about the decrementing function, there's clearly a flaw in my logic. And, in fact, it does not always terminate. So what do we

do about it?

Well this is the main trick. And one of the things I need you to understand this semester is perhaps the most important thing you'll learn is how to debug programs. Too many people think the hard part is done when they write the code the first time. No, that's actually the easy part. The hard part is actually getting it to work. So the thing you need to learn is how to debug code, and the nice thing is it's a transferable skill. It's also fine for debugging lab experiences or family members or anything else that can be seriously awry.

So the way I do it when I program is typically with print statements. So the fact that the program was running forever, suggests that I'm not exiting the while loop. So clearly, I need to print something in the while loop. And the only three variables it seems to be using are answer, low, and high. Those are the three that change. So let's see what the value is.

Notice, by the way, that I actually went to the trouble to type ans equals ans, low equals low, et cetera. A lot of people would just say, OK I'm going to print and they'll ans comma low comma high. And then when they run the code they'll forget which is which.

The most common problem that people have in debugging programs is that they are lazy. They think they're saving themselves work, and in fact they're creating work. So my first piece of advice to you as debuggers is don't be lazy. Maybe you heard this from your mother at some point in life, or your father. But now you're hearing it from me. Try and just do it right the first time.

So let's run it and see what happens now. Well at least it's printing some output. And it's chugging away and chug-- uh-oh-- so now, we see we have a real problem. We've reached a fixed point, where every time through the loop, nothing is changing. Well the first time we go through the loop and nothing changes we know we're in trouble, because nothing's going to ever change. And therefore, we're going to be in the loop forever.

So we see, I've gone to a stage where everything equals 0.5. And now if I go back and look at the code, and I ask myself the question well what happens when everything is 0.5, and I can see the problem. It's this statement here. Yep, turn it around maybe. But that's never going to change now, because it's 0.5 plus 0.5, divided by 2 is 0.5. And it'll just stay there forever. So that's my problem.

I have to somehow change my code now, so that this doesn't happen. So what is the problem? Now somebody can tell me, simply. What is the problem here? What was the flaw in my reasoning when I first set this program up? Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Louder please?

AUDIENCE: You don't have a high minus low is less than or equal to epsilon.

PROFESSOR: So the comment was I don't have a high minus low is less than or equal to epsilon. True, but that's not really the real flaw. Yeah?

AUDIENCE: The fraction is greater than the original fraction, so the solution is not in the search space.

PROFESSOR: Exactly. So the answer is the problem was that I did a search in a region, and the answer wasn't in that region. Because the square root of 0.5 does not lie between 0 and 0.5. Silly me, when I thought about it, I didn't think of finding the square root of numbers less than 1. So what's a simple fix?

Well what I can do is the following: I'll go back and say high is going to be the max of x and 1. So now I'm going to ensure that the square root actually does lie in the region I'm searching. I hope. Let's run it. Ah. All right, well I got to some stuff at the end which you shouldn't worry about, but it found something that I guess is a good enough answer. We'll get rid of that code I put in this morning which we'll get to this later.

OK so I've now fixed the program. Everyone with me on that? Any questions? And the thing to understand is conceptually what was wrong with my reasoning, that I'm

doing a search in a region where the answer doesn't lie. So I'm not going to find it. And the other thing to understand is my systematic way of finding the bug. Now I confess I knew the bug was there when I wrote the code, so I kind of cheated with the debugging. But even if I hadn't known, this is what I would have done. I would have put in that print statement.

All right, so now we have actually a pretty good piece of code for finding square roots. And as we looked at on Tuesday, I can use the same piece of code. I can modify it to get cube roots, or fourth roots, or fifth roots. And so I have a general framework for doing things. But it's pretty unsatisfying in that sense, because let's look at it.

If I wanted to find the square root of some number other than 0.5, I have to go and edit the code, replace the assignment to x by whatever I'm trying to do. If I want to do cube roots I have to cut and paste and edit and do things. There's no very good way to now embed this piece of code inside a larger computation. Imagine that I've got some 10,000 line program that needs to find the square root six or seven times, well now I'm going to have six or seven copies of this code in my program, for every time I need the square root. Clearly not what you want to do.

In general, having more code is a bad thing. So it's not like you're given an essay to write and someone tells you it's got to be 5,000 words, and you just sweat blood trying to figure out how to stretch it to be that long. In code, it's the other way around. Most of the time we want to make it shorter not longer. And the reason we want to do that is the difficulty of getting code to work grows, maybe even grows quadratically, or worse but the size of the code. So the more code you have, the harder it is to get it to work.

So one of the things good programmers learn to do is write less code. And so we don't measure productivity of a programmer by the number of lines of code they produce each day, but we measure it by the amount of functionality they produce each day. And we give them bonus points if they achieve the desired functionality with less code. So let's talk about how we can write less code and accomplish more.

Well to do that, we're going to look at a new language mechanism-- actually not new, but new to this class-- called a function. But before we do that, I want to pull back and talk about what it is we hope to accomplish by introducing functions into our programming language. We want to provide a mechanism that provides for two things: decomposition and abstraction.

What decomposition does, is it creates structure. It allows us to break our program up into something called modules. And the module we'll focus on today is function, but later we'll see there's another important kind of module in Python called the class. And the advantage of a module is it should be self-contained and reusable. So it's a self-contained unit of functionality that can be used in multiple contexts.

Abstraction suppresses details. It allows us to use a piece of code as if it were a black box. That is, something whose interior details we can't see, don't need to see, and shouldn't even want to see. We only need to understand what it does, not how it does it. And that lets us use code that other people have written easily. And, in fact, use code that we have written easily. It's one of those few occasions where I think Thomas Gray was right, when he said, "ignorance is bliss." Sometimes knowing less is better.

All right, so let's look at the way functions work. The functions let us break code into reusable, coherent pieces. Now we've already looked at similar kinds of things. When we looked at say floating point numbers, and we wrote operations like plus or divide, whatever, we didn't worry about how they were actually implemented in the machine.

We said OK they do something, they're kind of like dividing real numbers, let's not worry about the details. We do that with a lot of things. We looked at strings. We concatenated strings. Well we didn't worry about how did Python go about doing that. We just assumed it did it, and it had the meaning we wanted it to. What functions let us do is extend the language in some sense by adding new primitives that we can use just the way we used the built-in primitives.

So let's look at an example here. I've written a very simple function. Does something

that we actually did already when we looked at square roots. It's a function called within epsilon. And let me comment this out while I'm thinking about it, so we don't have to live with it later. And I now want to walk you, slowly, through what this function does.

So at the start, it uses the keyword Def, short for define. Following that is a name. I chose the name within epsilon. You can choose any name you want for a function. I'm strongly encourage you to choose mnemonic names, that is to say names that have a meaning. So in some sense you see it says within epsilon, and you know what it does already.

Following that, it has three things called formal parameters. I'll come back to in a minute, what that means. And then after that, it's got something called the function body. So we see that a function has a name, it has parameters, and it has a body.

The body is the code that's part of the function. In the body, you can write any code you want. Plus, there's something you can't write outside of a function, called return. That's a special command that says whoever calls me has called me to have me compute a value. I'm going to return the value that this person would want.

And then here we see something that's very important. This is where we get abstraction, and that's the specification of the function. And it says here, there are two pieces to it. One that its parameters x, y, and epsilon, are all floats. And furthermore, epsilon is greater than 0. You can imagine this is important, and it returns true if x is within epsilon of y. Otherwise it will return false.

If I want to use within epsilon, I don't need to look at the code. I look instead at the specification. Now here where the code is one line, maybe I haven't gained a lot by looking at the specification instead of the code. But you can imagine if the code were 1,000 lines, I'd much rather read the specification than the code. We'll also see for other reasons later why it's in fact dangerous to look at the code.

How do I use it? I use it by invoking it. So I could, for example write something like print within epsilon, of two, three, one. What's it going to print? Pardon? Why is it

going to print an error do you think?

AUDIENCE: Because you haven't put epsilon.

PROFESSOR: Ah, typed it wrong. Thank you. You're correct. It would have printed an error. Now what will it print?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Sure enough. I could also, if I chose, write something like val equals that, then if I want I could print val. Now it's going to print false. So within epsilon is just like plus or something else, does some computation, returns a value. I can use that value any place I could have used an expression.

Now one more thing to look at with this. Suppose I don't return anything. Anyone want to guess what it's going to do now? I point this out, because this is a very common error. People write lots of code, calculate some wonderful value and then forget to return it.

What's it going to do now? Well let's run it and see. That, by the way, is a good habit to get into. It's going to return the special value none. Remember we looked at that earlier, meaning I don't have a value. So if you see in your code some none popping up where you don't expect it to, it's probably because you forgot to return a value. So just keep that in mind.

All right, now there's a big advantage of this. Once I've written this code I can now anywhere I want call within epsilon, and I don't have to duplicate the code. I only do it once. As I said earlier maybe I'm not gaining much, because the body is so short. On the other hand, I'm still gaining something. Notice that when I look at the code down here, it's easy to read, I'm printing within epsilon two, three, and one.

And I don't have to decode this and tell me that that's what that's doing. So if I have a function and I choose the names properly, code that uses the function is much easier to read. And that can be a big value.

All right, let's look at another example. So here I've got this function, f. I've chose a

non-mnemonic name, because there isn't much meaning to this function. What f does, it is a formal parameter x. It sets x to x plus 1. Then it prints x and returns x-- pretty boring. So let's see what it does here.

So now I'm going to set x to three, set z, or zed if you happen to be Canadian, to f of x. And then print the values of z and x. Also in f, before I return x, I'm going to print it. So let's see what happens when I run this one. It prints four, four, and then three. All right, what's going on? Why did it do that?

Well it's pretty easy to see why it printed four here, because I called f of x with an x equal to 3, and then I incremented it by one, and then I printed it. It's probably also easy to understand why z was four, because I returned the value of x here, which was four and it printed it. But why is this x three?

And the answer is this x and that x have nothing to do with each other. Right? I could just as easily have chosen some other value for the formal parameter, say George, and said George is equal to George plus 1, print George, return George. There is no relation between the name of the formal and, in this case, x defined in the calling environment.

So now let's think about that by working slowly and carefully through what happens when we call a function. So the first thing that happens at the call, and I'll just work it through this one, is the formal parameter, x in this case, is bound-- and I'll come back to what binding means, that's a critical concept here-- to the value of the actual parameter. So these are important terms, actual and formal, which in this case, also happens to be called x.

But what's happening here, is upon entry of a function, a new scope is created. What's a scope? A scope is a mapping from names to objects. So if we look at what's going on over here, we can draw a little picture.

Well before I draw a picture, I'm going to look at a slightly more complicated example. Well, yeah let's do that. This one is not in your handout, but it is illustrative of, I think, what's really going on here. Here I've got another beautifully named

function, in this case f1, and inside it, I've defined another function, called g, which takes no arguments. I've set x to abc.

Then I haven't shown you these assert statements yet, or haven't talked about them. Assert is a command in which the keyword assert is followed by an expression that evaluates to either true or false. If it evaluates to true, it does nothing. It just continues. If it evaluates to false, it stops your program dead in its tracks. So I've just used it here as a trick to make my program stop when I run it. In general, you'll find that I use asserts quite a lot.

So for example, in the next piece of code, which is called find root. It takes the root, is it square, or cube, whatever, the value, and epsilon. It assumes that powers, and int, and val, and epsilon float, in the specification. And then you'll notice, I start by putting in an assertion here. And what I'm asserting is that the actuals to which these formals are bound, have the properties the specification says they do.

This is what's called defensive programming. In principle, I shouldn't have to do that, because in principle, nobody should call this with incorrect values. But, in fact, it can happen. Programmers occasionally make mistakes. And so I'm protecting myself by checking that the assumptions are met, and if they're not, my program will just stop. Then I can go hunt down the fool that called it with the wrong parameters-- probably myself.

So asserts are good for that, and I'll use them a lot for these kinds of things. I'll also use them when I think I know what value something should be in a program at some point, and I'm not sure it really is. I'll assert that it has the value I think it is. I'll assert that x is six, if I think it's going to be six. And then my program will conveniently stop for me if it's not true.

All right so that's assert. Other than that, I think there's nothing here you haven't seen before. So what's going to happen, we're going to step through this piece by piece. So initially, as we look at it, we enter the main body of the program, which is not wrapped in a function.

So what IDLE will do, or the interpreter will do, is it will start by executing each def. But executing a def doesn't do anything, but put some names in the environment. Then it will go and start actually running and interpreting the code that's not nested inside a function.

So the first thing that will happen is the interpreter will build for me what's called the scope. I've already mentioned, that's a mapping from names to objects. So in the outermost scope, it will first find the name f1. F1 it will tell me that f1 maps to an object that happens to be a function. So it will come over here-- and I'm just going to draw some picture, we'll assume that's the memory of the computer-- and it will map to something that happens to be a bunch of code, if you will. All right?

It will then stop. It will then notice that it's got, at the outermost level a variable called x. And that will map to an integer, which will initially have no value in it. And then after the assignment, it will now be bound to the object three. It will then create another object z, but before it can bind a value to it, it will invoke the function f1.

Now the interpreter starts to execute f1. When it does that, it will create another scope. So this is the main scope. It will next create a scope called the f1 scope. In that, it will have another name g, which will be bound to some code. It will have a name x, which will be initially bound to the actual. So in this case, it will be bound to the object three. We'll then eventually do a print.

It will involve g, which will now create the g scope. And the g scope will create a name x, which in this case will be bound to the string abc. It will then start executing g, and it will stop. So let's see what that looks like.

Sure enough, it got an assert false, gave an assertion error. What I can do now is go up to this debug here, and go to what's called a stack viewer. Each of these scopes is what's called a stack frame.

Now why are they called stack frames? Because when we do it, we begin with the main scope. We call f, and we get the scope. f calls g and we get the g scope. When g completes, which alas it doesn't because of the error, it pops the stack, and gets

rid of the g scope. And now the stack is the f and the main, and then when f completes, it will have just the main. So it's last in, first out, which is typically called a stack in computing-- or a LIFO, if you're a course 15 major, and do accounting.

So let's look at the stack viewer. And I apologize for the small type font, but I was unable to make it look bigger. So it says at the top we've got an assertion error. And then you'll note it's got three stacks: the main, the f1 stack, and the g stack. I forgot I called it f1, not f. Then I can inspect them further.

So the g stack has local and global variables. The local variables include x, which is equal to abc. Globals we'll get to later. If I look at f1, it also has a local called x, but its value is now four, not abc. And it has a value called g, which is a function, as we discussed. And if I look at main, it has a bunch of things, but it has everything that's available in the interpreter, which because we've looked at within epsilon it's there. But you'll notice it's x is 3. All right?

So the stack viewer can be very handy, to look at what you've got, when you've got a bunch of calls. Now if we go back to our code here, and we'll take this out. And suppose what we do is we assert false here. Now if we look at the stack viewer, we see that we have f1 in main, but g is no longer there. It's gone. All those variables don't exist anymore, because I'm no longer in g.

This is the nice thing, because it means if you call something 1,000 times, it doesn't use up all your memory. Every time it's finished, it gets rid of what it no longer needs. All right? Does that make sense? This is an important thing to get-- yeah, thank you, question.

AUDIENCE: --the assertion.

PROFESSOR: Where did I--

AUDIENCE: Where did you put this other assertion, when you just changed--

PROFESSOR: Ah, where did I put the other assertion? If we look at the code, you'll see I put it after I call g, and g is by now returned, but before I left f1, which is why the f1 stack is still

present. That makes sense to you? Which stacks exist, which stack frames exist, depends upon which functions are still active. Yeah?

AUDIENCE: How come you don't need a return under the g function?

PROFESSOR: Oh, because there's not going to be anything interesting. It's useless. Right? Why don't I need a return under g? Well if I wanted it to do something useful, I would need to return something. But I'd probably also want to pass it some arguments, rather than have it take no arguments, as well. So it's here just to be the simplest thing I could put that created a stack frame. But don't try and interpre it as being anything meaningful. Yeah?

AUDIENCE: Would you run into problems assuming that g did something to x and then returned it? Would you run into any problems that you named the variable the same? You know, that you used x twice? Would you want to--

PROFESSOR: No. If an x exists, or any variable exists within a function body, when you leave that function, that variable is gone forever. These are just names. They have no intrinsic meaning. So one of the ways to think about it, and we'll see this later when we get to classes-- a lot later. You could, if you wanted, think about this as really the name g.x, and you could really think of this as the name of f1.x, and you could think of this as the name of main.x, indicating that they're really not the same. But it would be kind of a pain to write them all that way. OK?

So different scopes have different names available to them. You can use the names in the scope, and you have to keep track of what they mean. OK? Any other questions? These are great questions, and I really do appreciate them. Yeah?

AUDIENCE: Does this also happen with four loops? Like if you say 4x in range something, can you use x later? Or is it x--

PROFESSOR: You can use x later.

AUDIENCE: Okay so--

PROFESSOR: x will be available outside the loop. This was if you said for x in something, is x

available outside the loop? Yes. And in fact, you'll often want to test what the final value of x is, when you leave the loop.

OK, the next thing on your handout, and I'm not going to go over it, is using functions to implement something that finds roots. There's no real point in my walking you through this code in class. I did include it in the handout. And by the way, the handouts are all available after lecture, online. Is that, I think you should work through in your own, and make sure you understand it, to get a sense of how functions work.

And it's certainly related to the current problem set, which would be another good reason to work through it-- the problem set that will be posted today-- the new problem set, PS 2.

Note again how careful I am about the specifications. And I should point out something interesting, if I type find root, open para-- well let's do this here. Let's clear things up. Let's get rid of things that will cause the program to halt.

Notice that when I type find root open, open paren, it's given me the values-- the names of the formal parameters, which I've chosen in such way that will remind me what their value should be. And it's also given me part of the specification, the piece in the triple quotation marks to tell me the rules I'm supposed to be following here on these things. So it's a very handy thing. And as you use IDLE, you'll get used to the fact that this is a convenience.

All right, work your way through that code. Make sure you know what it does. Finally, today I want to switch gears, and talk about something else. Up till now, all of the programs we've looked at have been numeric-- they've played with numbers. And I've done that, because I assumed you guys all had some intuition about numbers.

I've use strings as a primitive data element to print things, but we haven't done anything very interesting with strings. However strings are indeed quite interesting, in that they're the first non-scalar value we've looked at. You'll recall non-scalar

values are values that can be decomposed.

So if we now look at the code again, I've got this little piece of code called sumDigits. So before, the for statement we looked at was for x in range. Well you can apply it to a for statement to any type that has a way to enumerate its elements.

So for c in STR, actually of 1952, so I've taken the number 1952 and converted it to a string so it will now be quote one nine five two, I can now do something to every character in that string. And what I'm doing is converting it back to an int, and then adding it. So this will give me the sum of the digits. 17. This is a very convenient mechanism, and you'll use for a lot, this way. You'll use it in fact more for this sort of thing then you will for ints.

Now I can also select values. So if I look at-- I don't know what's going on here. Every once in while when you go back and forth between the editor and the shell, the shell hangs and you have to go try it again. If I go to s equals abc, I can look at individual elements of s, for example s sub 0, which will be a. I can also look at slices of s. So for example s from 0 to one.

That's interesting. What is it doing? Now try and infer. I'll give you another example.

So you'll remember when we did range from x to y, it went y minus 1. Same kind of thing is happening here. So that's why s from 0 to one gives me only one character, but s from 0 to two gives me the character string ab. This is what's called slicing, and it's very common. What a slice does, is it makes a new copy-- makes a new object, in this case-- which is a sub-string of the original string.

There are many other things I can do on strings. I can do something like s.find, and it will tell me that b is at position number one in s. So use Google, whatever you use, to find the Python web page that describe strings, and it will give you all of the operations you can do. And they're quite convenient. One other scalar type that you're going to need for the problem set is tuples, and that will be discussed in recitation tomorrow.

OK, thanks a lot.