PROFESSOR JOHN GUTTAG: All right. That said, let's continue, and if you remember last time, we ended up looking at this thing I called square roots bi. This was using something called a bisection method, which is related to something called binary search, which we'll see lots more of later, to find square roots.

And the basic idea was that we had some sort of a line, and we knew the answer was somewhere between this point and this point. The line is totally ordered. And what that means, is that anything here is smaller than anything to its right. So the integers are totally ordered, the reals are totally ordered, lots of things are, the rationals are totally ordered.

And that idea was, we make a guess in the middle, we test it so this is kind of a guess and check, and if the answer was too big, then we knew that we should be looking over here. If it was too small, we knew we should be looking over here, and then we would repeat. So this is very similar, this is a kind of recursive thinking we talked about earlier, where we take our problem and we make it smaller, we solve a smaller problem, et cetera.

All right. So now, we've got it, I've got the code up for you. I want you to notice the specifications to start. We're assuming that x is greater than or equal to 0, and epsilon is strictly greater than 0, and we're going to return some value y such that y squared is within epsilon of x.

I'd last time talked about the two assert statements. In some sense, strictly speaking they shouldn't be necessary, because the fact that my specification starts with an assumption, says, hey you, who might call square root, make sure that the things you call me with obey the assumption.

On the other hand, as I said, never trust a programmer to do the right thing, so we're going to check it. And just in case the assumptions are not true, we're just going to stop dead in our tracks.

All right. Then we're going to set low to-- low and high, and we're going to perform exactly the process I talked about. And along the way, I'm keeping track of how many iterations, at the end I'll print how many iterations I took, before I return the final guess.

All right, let's test it. So one of the things I want you to observe here, is that instead of sitting there and typing away a bunch of test cases, I took the trouble to write a function, called test bi in this case. All right, so what that's doing, is it's taking the things I would normally type, and putting them in a function, which I can then call.

Why is that better than typing them? Why was it worth creating a function to do this? Pardon?

STUDENT:: [INAUDIBLE]

PROFESSOR JOHN GUTTAG: Then I can I can use it again and again and again. Exactly.

By putting it in a function, if I find a bug and I change my program, I can just run the function again. The beauty of this is, it keeps me from getting lazy, and not only testing my program and the thing that found the bug, but in all the things that used to work.

We'll talk more about this later, but it often happens that when you change your program to solve one problem, you break it, and things that used to work don't work. And so what you want to do, and again we'll come back to this later in the term, is something called regression testing. This has nothing to do with linear regression. And that's basically trying to make sure our program has not regressed, as to say, gone backwards in how well it works. And so we always test it on everything.

All right? So I've created this function, let's give it a shot and see what happens. We'll run test bi. Whoops!

All right, well let's look at our answers. I first tested it on the square root of 4, and in one iteration it found 2. I like that answer. I then tested it on the square root of 9, and as I mentioned last time, I didn't find 3. I was not crushed. You know, I was not really disappointed, it found something close enough to 3 that I'm happy.

All right. I tried it on 2, I surely didn't expect a precise and exact answer to that, but I got something, and if you square this, you'll find the answer kept pretty darn close to 2.

I then tried it on 0.25 One quarter. And what happened was not what I wanted. As you'll see, it crashed.

It didn't really crash, it found an assert statement. So if you look at the bottom of the function, you'll see that, in fact, I checked for that. I assert the counter is less than or equal to 0. I'm checking that I didn't leave my program because I didn't find an answer. Well, this is a good thing, it's better than my program running forever, but it's a bad thing because I don't have it the square root of 0.25.

What went wrong here? Well, let's think about it for a second. You look like-- someone looks like they're dying to give an answer. No, you just scratching your head? All right.

Remember, I said when we do a bisection method, we're assuming the answer lies somewhere between the lower bound and the upper bound. Well, what is the square root of a quarter? It is a half.

Well, what-- where did I tell my program to look for an answer? Between 0 and x. So the problem was, the answer was over here somewhere, and so I'm never going to find it cleverly searching in this region, right? So the basic idea was fine, but I failed to satisfy the initial condition that the answer had to be between the lower bound and the upper bound. Right?

And why did I do that? Because I forgot what happens when you look at fractions. So what should I do? Actually I lied, by the way, when I said the answer was over there. Where was the answer? Somebody?

It was over here. Because the square root of a quarter is not smaller than a quarter, it's bigger than a quarter. Right? A half is strictly greater than a quarter.

So it wasn't on the region. So how-- what's the fix? Should be a pretty simple fix, in fact we should be able to do it on the fly, here. What should I change? Do I need to change the lower bound? Is the square root ever going to be less than 0? Doesn't need to be, so, what should I do about the upper bound here? Oh, I could cheat and make, OK, the upper bound a half, but that wouldn't be very honest.

What would be a good thing to do here? Pardon? I could square x, but maybe I should just do something pretty simple here. Suppose-- whoops. Suppose I make it the max of x and 1. Then if I'm looking for the square root of something less than 1, I know it will be in my region, right?

All right, let's save this, and run it and see what happens. Sure enough, it worked and, did we get-- we got the right answer, 0.5 All right? And by the way, I checked all of my previous ones, and they work too.

All right. Any questions about bisection search?

One of the things I want you to notice here is the number iterations is certainly not constant. Yeah, when I will looked at 4, it was a nice number like 1, 9 looked like it took me 18, 2 took me 14, if we try some big numbers it might take even longer. These numbers are small, but sometimes when we look at really harder problems, we got ourselves in a position where we do care about the number of iterations, and we care about something called the speed of convergence.

Bisection methods were known to the ancient Greeks, and it is believed by many, even to the Babylonians. And as I mentioned last time, this was the state of the art until the 17th century. At which point, things got better. So, let's think about it, and let's think about what we're actually doing when we solve this.

When we look for something like the square root of x, what we're really doing, is solving an equation. We're looking at the equation f of guess equals the guess squared minus x. Right, that's what that is equal to, and we're trying to solve the equation that f of guess equals 0. Looking for the root of this equation.

So if we looked at it pictorially, what we've got here is, we're looking at f of x, I've plotted it here, and we're asking where it crosses the x axis. Sorry for the overloading of the word x.

And I'm looking here at 16. Square root of 16, and my plot basically shows it crosses at 4 and-- well, I think that's minus 4. The perspective is tricky-- and so we're trying to find the roots.

Now Isaac Newton and/or Joseph Raphson figured out how to do this kind of thing for all differentiable functions. Don't worry about what that means.

The basic idea is, you take a guess, and you -- whoops -- and you find the tangent of that guess.

So let's say I guessed 3. I look for the tangent of the curve at 3. All right, so I've got the tangent, and then my next guess is going to be where the tangent crosses the x axis. So instead of dividing it in half, I'm using a different method to find the next guess.

The utility of this relies upon the observation that, most of the time-- and I want to emphasize this, most of the time, that implies not all of the time-- the tangent line is a good approximation to the curve for values near the solution. And therefore, the x intercept of the tangent will be closer to the right answer than the current guess.

Is that always true, by the way? Show me a place where that's not true, where the tangent line will be really bad. Yeah. Suppose I choose it right down there, I guess 0. Well, the tangent there will not even have an x intercept. So I'm really going to be dead in the water.

This is the sort of thing that people who do numerical programming worry about all the time. And there are a lot of a little tricks they use to deal with that, they'll perturb it a little bit, things like that. You should not, at this point, be worrying about those things.

This method, interestingly enough, is actually the method used in most hand calculators. So if you've got a calculator that has a square root button, it's actually in the calculator running Newton's method. Now I know you thought it was going to do that thing you learned in high school for finding square roots, which I never could quite understand, but no. It uses Newton's method to do it.

So how do we find the intercept of the tangent, the x intercept? Well this is where derivatives come in. What we know is that the slope of the tangent is given by the first derivative of the function f at the point of the guess. So the slope of the guess is the first derivative. Right. Which dy over dx. Change in y divided by change in x.

So we can use some algebra, which I won't go through here, and what we would find is that for square root, the derivative, written f prime of the i'th guess is equal to two times the i'th guess. Well, should have left myself a little more room, sorry about that.

All right? You could work this out. Right? The derivative of the square root is not a complicated thing. Therefore, and here's the key thing we need to keep in mind, we'll know that we can choose guess i plus 1 to be equal to the old guess, guess i, minus whatever the value is of the new guess-- of the old rather, the old guess-- divided by twice the old guess.

All right, again this is straightforward kind of algebraic manipulations to get here. So let's look at an example.

Suppose we start looking for the square root of 16 with the guess 3. What's the value of the function f of 3? Well, it's going to be, we looked at our function there, guess squared, 3 times 3 is 9 I think, minus 16, that's what x is in this case, which equals minus 7.

That being the case, what's my next guess? Well I start with my old guess, 3, minus f of my old guess, which is minus 7, divided by twice my old guess, which is 6, minus the minus, and I get as my new guess 4.1666 or thereabouts. So you can see I've missed, but I am closer. And then I would reiterate this process using that as guess i, and do it again.

One way to think about this intuitively, if the derivative is very large, the function is changing quickly, and therefore we want to take small steps. All right. If the derivative is small, it's not changing, maybe want to take a larger step, but let's not worry about that, all right?

Does this method work all the time? Well, we already saw no, if my initial guess is zero, I don't get anywhere. In fact, my program crashes because I end up trying to divide by zero, a really bad thing. Hint: if you implement Newton's method, do not make your first guess zero.

All right, so let's look at the code for that. All right so-- yeah, how do I get to the code for that? That's interesting.

All right. So we have that square root NR. NR for Newton Raphson. First thing I want you to observe is its specification is identical to the specification of square root bi. What that's telling me is that if you're a user of this, you don't care how it's implemented, you care what it does. And therefore, it's fine that the specifications are identical, in fact it's a good thing, so that means if someday Professor Grimson invents something that's better than Newton Raphson, we can all re-implement our square root functions and none of the programs that use it will have to change, as long as the specification is the same.

All right, so, not much to see about this. As I said, the specifications is the same, same assertions, and the-- it's basically the same program as the one we were just looking at, but I'm starting with a different guess, in this case x over 2, well I'm going to, couple of different guesses we can start with, we can experiment with different guesses and see whether we get the same answer, and in fact, if we did, we would see we didn't get this, we got different answers, but correct answers. Actually now, we'll just comment that out. I'm going to compute the difference, just as I did on the board, and off we'll go.

All right. Now, let's try and compare these things. And what we're going to look at is another procedure, you have the code for these things on your handout so we won't worry, don't need to show you the code, but let's look at how we're going to test it.

I'm doing a little trick by the way, I'm using raw input in my function here, as a just a way to stop the display. This way I can torture you between tests by asking you questions. Making it stop.

All right, so, we'll try some things. We'll see what it does. Starting with that, well, let's look at some of the things it will do. Yeah, I'll save it.. It's a little bit annoying, but it makes the font bigger.

All right, so we've tested it, and we haven't tested it yet, we have tested it but, we haven't seen it, well, you know what I'm going to do? I'm going to tort-- I'm going to make the font smaller so we can see more. Sorry about this. Those of you in the back, feel free to move forward.

All right. So we've got it, now let's test it. So we're going to do here, we're going to run compare methods. Well we're seeing this famous computers are no damn good.

All right. So we're going to try it on 2, and at least we'll notice for 2, that the bisection method took eight iterations, the Newton Raphson only took three, so it was more efficient. They came up with slightly different answers, but both answers are within .01 which is what I gave it here for epsilon, so we're OK. So even though they have different answers, they both satisfy the same specification, so we have no problem. All right?

Try it again, just for fun. I gave it here a different epsilon, and you'll note, we get different answers. Again, that's OK. Notice here, when I asked for a more precise answer, bisection took a lot more iterations, but Newton Raphson took only one extra iteration to get that extra precision in the answer. So we're sort of getting the notion that Newton Raphson maybe is considerably better on harder problems. Which, by the way, it is.

We'll make it an even harder problem, by making it looking an even smaller epsilon, and again, what you'll see is, Newton Raphson just crept up by one, didn't take it long, and got the better answer, where bisection gets worse and worse. So as you can see, as we escalate the problem difficulty, the difference between the good method and the not quite as good method gets bigger and bigger and bigger. That's an important observation, and as we get to the part of the course, we talk about computational complexity, you'll see that what we really care about is not how efficient the program is on easy problems, but how efficient it is on hard problems.

All right. Look at another example. All right, here I gave it a big number, 123456789. And again, I don't want to bore you, but you can see what's going on here with this trend.

So here's an interesting question. You may notice that it's always printing out the same number of digits. Why should this be? If you look at it here, what's going on? Something very peculiar is happening here. We're looking at it, and we're getting some funny answers.

This gets back to what I talked about before, about some of the precision of floating point numbers. And the thing I'm trying to drive home to you here is perhaps the most important lesson we'll talk about all semester. Which is, answers can be wrong.

People tend to think, because the computer says it's so, it must be so. That the computer is-- speaks for God. And therefore it's infallible. Maybe it speaks for the Pope. It speaks for something that's infallible. But in fact, it is not. And so, something I find myself repeating over and over again to myself, to my graduate students, is, when you get an answer from the computer, always ask yourself, why do I believe it? Do I think it's the right answer? Because it isn't necessarily.

So if we look at what we've got here, we've got something rather peculiar, right? What's peculiar about what this computer is now printing for us? Why should I be really suspicious about what I see in the screen here?

STUDENT: [INAUDIBLE]

PROFESSOR JOHN GUTTAG: Well, not only is it different, it's really different, right? If it were just a little bit different, I could say, all right, I have a different approximation. But when it's this different, something is wrong. Right?

We'll, later in the term when we get to more detailed numerical things, look at what's wrong. You can run into issues of things like overflow, underflow, with floating point numbers, and when you see a whole bunches of ones, it's particularly a good time to be suspicious. Anyway the only point I'm making here is, paranoia is a healthy human trait.

All right. We can look at some other things which will work better. And we'll now move on. OK.

So we've looked at how to solve square root we've, looked at two problems, I've tried to instill in you this sense of paranoia which is so valuable, and now we're going to pull back and return to something much simpler than numbers, and that's Python. All right? Numbers are hard. That's why we teach whole semesters worth of courses in number theory. Python it's easy, which is why we do it in about four weeks.

All right. I want to return to some non-scalar types. So we've been looking, the last couple of lectures, at floating point numbers and integers. We've looked so far really at two non-scalar types. And those were tuples written with parentheses, and strings.

The key thing about both of them is that they were immutable. And I responded to at least one email about this issue, someone quite correctly said tuple are immutable, how can I change one? My answer is, you can't change one, but you can create a new one that is almost like the old one but different in a little bit.

Well now we're going to talk about some mutable types. Things you can change. And we're going to start with one that you, many of you, have already bumped into, perhaps by accident, which are lists. Lists differ from strings in two ways; one way is that it's mutable, the other way is that the values need not be characters. They can be numbers, they can be characters, they can be strings, they can even be other lists.

So let's look at some examples here. What we'll do, is we'll work on two boards at once. So I could write a statement like, techs, a variable, is equal to the list, written with the square brace, not a parenthesis, MIT, Cal Tech, closed brace. What that basically does, is it takes the variable techs, and it now makes it point to a list with two items in it. One is the string MIT and one is the string Cal Tech.

So let's look at it. And we'll now run another little test program, show lists, and I printed it, and it prints the list MIT, Cal Tech. Now suppose I introduce a new variable, we'll call it ivys, and we say that is equal to the list Harvard, Yale, Brown. Three of the ivy league colleges. What that does is, I have a new variable, ivys, and it's now pointing to another, what we call object, in Python and Java, and many other languages, think of these things that are sitting there in memory somewhere as objects. And I won't write it all out, I'll just write it's got Harvard as one in it, and then it's got Yale, and then it's got Brown. And I can now print ivys. And it sure enough prints what we expected it to print.

Now, let's say I have univs, for universities, equals the empty list. That would create something over here called univs, another variable, and it will point to the list, an object that contains nothing in it.

This is not the same as none. It's it does have a value, it just happens to be the list that has nothing in it.

And the next thing I'm going to write is univs dot append tex. What is this going to do? It's going to take this list and add to it something else.

Let's look at the code. I'm going to print it, and let's see what it prints. It's kind of interesting. Whoops. Why did it do that? That's not what I expected. It's going to print that. The reason it printed that is I accidentally had my finger on the control key, which said print the last thing you had.

Why does it start with square braced square brace? I take it-- yes, go ahead.

STUDENT: So you're adding a list to a list?

PROFESSOR JOHN GUTTAG: So I'm adding a list to a list. What have I-- what I've appended to the empty list is not the elements MIT and Cal Tech but the list that contains those elements.

So I've appended this whole object. Since that object is itself a list, what I get is a list of lists.

Now I should mention this notation here append is what is in Python called a method. Now we'll hear lots more about methods when we get to classes and inheritance, but really, a method is just a fancy word for a function with different syntax. Think of this as a function that takes two arguments, the first of which is univs and the second of which is techs. And this is just a different syntax for writing that function call.

Later in the term, we'll see why we have this syntax and why it wasn't just a totally arbitrary brain-dead decision by the designers of Python, and many languages before Python, but in fact is a pretty sensible thing. But for now, think of this as just another way to write a function call. All right, people with me so far?

Now let's say we wanted as the next thing we'll do, is we're going to append the ivys to univ. Stick another list on it. All right. So we'll do that, and now we get MIT, Cal Tech, followed by that list followed by the list Harvard, Yale, Brown. So now we have a list containing two lists.

What are we going to try next? Well just to see what we know what we're doing, let's look at this code here. I've written a little for loop, which is going to iterate over all of the elements in the list. So remember, before we wrote things like for i in range 10, which iterated over a list or tuple of numbers, here you can iterate over any list, and so we're going to just going to take the list called univs and iterate over it.

So the first thing we'll do is, we'll print the element, in this case it will be a list, right? Because it's a list with two lists in it. Then the next thing in the loop, we're going to enter a nested loop, and say for every college in the list e, we're going to print the name of the college. So now if we look what we get-- do you not want to try and execute that?-- it'll first print the list containing MIT and Cal Tech, and then separately the strings MIT and Cal Tech, and then the list containing Harvard, Yale, and Brown, and then the strings Harvard, Yale, and Brown.

So we're beginning to see this is a pretty powerful notion, these lists, and that we can do a lot of interesting things with them. Suppose I don't want all of this structure, and I want to do what's called flattening the list. Well I can do that by, instead of using the method append, use the concatenation operator. So I can concatenate techs plus ivys and assign that result to univs, and then when I print it you'll notice I just get a list of five strings.

So plus and append do very different things. Append sticks the list on the end of the list, append flattens it, one level of course. If I had lists of lists of lists, then it would only take out the first level of it. OK, very quiet here. Any questions about any of this? All right. Because we're about to get to the hard part Sigh. All right.

Let's look at the-- well, suppose I want to, quite understandably, eliminate Harvard. All right, I then get down here, where I'm going to remove it. So this is again another method, this is remove, takes two arguments, the first is ivys, the second is the string Harvard. It's going to search through the list until the first time it finds Harvard and then it's going to yank it away. So what happened here? Did I jump to the wrong place?

STUDENT: You hit two returns.

PROFESSOR JOHN GUTTAG: I hit two returns. Pardon?

STUDENT: You hit two returns. One was at

STUDENT: Pardo

PROFESSOR JOHN GUTTAG: This one.

STUDENT: No, up one.

PROFESSOR JOHN GUTTAG: Up one.

STUDENT: Right.

PROFESSOR JOHN GUTTAG: But why is Harvard there?

STUDENT: I'm sorry, I didn't write it down. PROFESSOR JOHN GUTTAG: Let's look at it again. All right, it's time to interrupt the world, and we'll just type into the shell. Let's see what we get here. All right, so let's just see what we got, we'll print univs. Nope, not defined.

All right, well let's do a list equals, and we'll put some interesting things in it, we'll put a number in it, because we can put a number, we'll put MIT in it, because we can put strings, we'll put another number in it, 3.3, because we can put floating points, we can put all sorts of things in this list. We can put a list in the list again, as we've seen before. So let's put the list containing the string a, and I'll print out, so now we see something pretty interesting about a list, that we can mix up all sorts of things in it, and that's OK.

You'll notice I have the string with the number 1, a string with MIT, and then it just a plain old number, not a string, again it didn't quite give me 3.3 for reasons we've talked before, and now it in the list a.

So, suppose I want to remove something. What should we try and remove from this list? Anybody want to vote? Pardon? All right, someone wants to remove MIT. Sad but true. Now what do we get if we print l? MIT is gone.

How do I talk about the different pieces of l? Well I can do this. l sub 0-- whoops-- will give me the first element of the list, just as we could do with strings, and I can look at l sub minus 1 to get the last element of the list, so I can do all the strings, all the things that I could do with strings.

It's extremely powerful, but what we haven't seen yet is mutation. Well, we have seen mutation, right? Because notice that what remove did, it was it actually changed the list. Didn't create a new list. The old l is still there, but it's different than it used to be. So this is very different from what we did with slicing, where we got a new copy of something. Here we took the old one and we just changed it.

On Thursday, we'll look at why that allows you to do lots of things more conveniently than you can do without mutation.