ANNOUNCER: Open content is provided under a creative commons license. Your support will help MIT OpenCourseWare continue to offer high-quality educational resources for free. To make a donation, or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu .

PROFESSOR JOHN GUTTAG: Good morning.

We should start with the confession, for those of you looking at this on OpenCourseWare, that I'm currently lecturing to an empty auditorium. The fifth lecture for 600 this term, we ran into some technical difficulties, which left us with a recording we weren't very satisfied with. So, I'm-- this is a redo, and if you will hear no questions from the audience and that's because there is no audience.

Nevertheless I will do my best to pretend. I've been told this is a little bit like giving a speech before the US Congress when C-SPAN is the only thing watching.

OK. Computers are supposed to be good for crunching numbers. And we've looked a little bit at numbers this term, but I now want to get into looking at them in more depth than we've been doing.

Python has two different kinds of numbers. So far, the only kind we've really paid any attention to is type int. And those were intended to mirror the integers, as we all learned about starting in elementary school. And they're good for things that you can count. Any place you'd use whole numbers.

Interestingly, Python, unlike some languages, has what are called arbitrary precision integers. By that, we mean, you can make numbers as big as you want them to.

Let's look at an example. We'll just take a for a variable name, and we'll set a to be two raised to the one-thousandth power. That, by the way, is a really big number. And now what happens if we try and display it? We get a lot of digits. You can see why I'm doing this on the screen instead of writing it on the blackboard.

I'm not going to ask you whether you believe this is the right answer, trust me, trust Python. I would like you to notice, at the very end of this is the letter L.

What does that mean? It means long. That's telling us that it's representing these-- this particular integer in what it calls it's internal long format.

You needn't worry about that. The only thing to say about it is, when you're dealing with long integers, it's a lot less efficient than when you're dealing with smaller numbers. And that's all it's kind of warning you, by printing this L.

About two billion is the magic number. When you get over two billion, it's now going to deal with long integers, so if, for example, you're trying to deal with the US budget deficit, you will need integers of type L.

OK. Let's look at another interesting example. Suppose I said, b equal to two raised to the nine hundred ninety-ninth power. I can display b, and it's a different number, considerably smaller, but again, ending in an L.

And now, what you think I'll get if we try a divided by b? And remember, we're now doing integer division. Well, let's see.

We get 2L. Well, you'd expect it to be two, because if you think about the meaning of exponentiation, indeed, the difference between raising something to the nine hundred ninety-ninth power and to the one-thousandth power should be, in this case, two, since that's what we're raising to a power.

Why does it say 2L, right? Two is considerably less than two billion, and that's because once you get L, you stay L. Not particularly important, but kind of worth knowing.

Well, why am I bothering you with this whole issue of how numbers are represented in the computer? In an ideal world, you would ignore this completely, and just say, numbers do what numbers are supposed to do.

But as we're about to see, sometimes in Python, and in fact in every programming language, things behave contrary to what your intuition suggests. And I want to spend a little time helping you understand why this happens. So let's look at a different kind of number.

And now we're going to look at what Python, and almost every other programming language, calls type float. Which is short for floating point. And that's the way that programming languages typically represent what we think of as real numbers.

So, let's look at an example. I'm going to set the variable x to be 0.1, 1/10, and now we're going to display x.

Huh? Take a look at this. Why isn't it .1? Why is it 0.1, a whole bunch of zeros, and then this mysterious one appearing at the end?

Is it because Python just wants to be obnoxious and is making life hard? No, it has to do with the way the numbers are represented inside the computer. Python, like almost every modern programming language, represents numbers using the i triple e floating point standard, and it's i triple e 754.

Never again will you have to remember that it's 754. I promise not to ask you that question on a quiz.

But that's what they do. This is a variant of scientific notation. Something you probably learned about in high

school, as a way to represent very large numbers.

Typically, the way we do that, is we represent the numbers in the form of a mantissa and an exponent. So we represent a floating point number as a pair, of a mantissa and an exponent.

And because computers work in the binary system, it's unlike what you probably learned in high school, where we raise ten to some power. Here we'll always be raising two to some power. Maybe a little later in the term, if we talk about computer architecture, we'll get around to explaining why computers working binary, but for now, just assume that they do and in fact they always have.

All right. Purists manage to refer to the mantissa as a significant, but I won't do that, because I'm an old guy and it was a mantissa when I first learned about it and I just can't break myself of the habit.

All right. So how does this work? Well, when we recognize so-- when we represent something, the mantissa is between one and two. Whoops. Strictly less than two, greater than or equal to one.

The exponent, is in the range, -1022 to +1023. So this lets us represent numbers up to about 10 to the 308th, plus or minus 10 to the 308th, plus or minus. So, quite a large range of numbers.

Where did these magic things come from? You know, what-- kind of a strange numbers to see here. Well, it has to do with the fact that computers typically have words in them, and the words today in a modern computer are 64 bits. For many years they were 32 bits, before that they were 16 bits, before that they were 8 bits, they've continually grown, but we've been at 64 for a while and I think we'll be stuck at 64 for a while.

So as we do this, what we do is, we get one bit for the sign-- is it a positive or negative number?-- 11 for the exponent, and that leaves 52 for the mantissa. And that basically tells us how we're storing numbers.

Hi, are you here for the 600 lecture? There is none today, because we have a quiz this evening. It's now the time that the lecture would normally have started, and a couple of students who forgot that we have a quiz this evening, instead of a lecture, just strolled in, and now strolled out.

OK. You may never need to know these constants again, but it's worth knowing that they exist, and that basically, this gives us about the equivalent of seventeen decimal digits of precision. So we can represent numbers up to seventeen decimal digits long. This is an important concept to understand, that unlike the long ints where they can grow arbitrarily big, when we're dealing with floating points, if we need something more than seventeen decimal digits, in Python at least, we won't be able to get it. And that's true in many languages.

Now the good news is, this is an enormous number, and it's highly unlikely that ever in your life, you will need

more precision than that.

All right. Now, let's go back to the 0.1 mystery that we started at, and ask ourselves, why we have a problem representing that number in the computer, hence, we get something funny out from we try and print it back.

Well, let's look at an easier problem first. Let's look at representing the fraction 1/8. That has a nice representation. That's equal in decimal to 0.125, and we can represent it conveniently in both base 10 and base 2.

So if you want to represent it in base 10, what is it? What is that equal to? Well, we'll take a mantissa, 1.25, and now we need to multiply it by something that we can represent nicely, and in fact that will be times 10 to the -1. So the exponent would simply be -1, and we have a nice representation.

Suppose we want to represent it in base 2? What would it be? 1.0 times-- anybody?-- Well, 2 to the -3. So, in binary notation, that would be written as 0.001.

So you see, 1/8 is kind of a nice number. We can represent it nicely in either base 10 or base 2.

But how about that pesky fraction 1/10? Well, in base 10, we know how to represent, it's 1 times 10 to the-- 10 to the what?-- 10 to the 1? No.

But in base 2, it's a problem. There is no finite binary number that exactly represents this decimal fraction. In fact, if we try and find the binary number, what we find is, we get an infinitely repeating series. Zero zero zero one one zero zero one one zero zero, and et cetera. Stop at any finite number of bits, and you get only an approximation to the decimal fraction 1/10.

So on most computers, if you were to print the decimal value of the binary approximation-- and that's what we're printing here, on this screen, right? We think in decimal, so Python quite nicely for us is printing things in decimal-- it would have to display-- well I'm not going to write it, it's a very long number, lots of digits-- however, in Python, whenever we display something, it uses the built-in function repr, short for representation, that it converts the internal representation in this case of a number, to a string, and then displays that string in this case on the screen. For floats, it rounds it to seventeen digits. There's that magic number seventeen again. Hence, when it rounds it to seventeen digits, we get exactly what you see in the bottom of the screen up there.

Answer to the mystery, why does it display this? Now why should we care? Well, it's not so much that we care about what gets displayed, but we have to think about the implications, at least sometimes we have to think about the implications, of what this inexact representation of numbers means when we start doing more-or-less complex computations on those numbers.

So let's look at a little example here. I'll start by starting the variable s to 0.0 . Notice I'm being careful to make it a float. And then for i in range, let's see, let's take 10, we'll increase s by 0.1 .

All right, we've done that, and now, what happens when I print s? Well, again you don't get what your intuition says you should get. Notice the last two digits, which are eight and nine. Well, what's happening here?

What's happened, is the error has accumulated. I had a small error when I started, but every time I added it, the error got bigger and it accumulates. Sometimes you can get in trouble in a computation because of that.

Now what happens, by the way, if I print s? That's kind of an interesting question.

Notice that it prints one. And why is that? It's because the print command has done a rounding here. It automatically rounds.

And that's kind of good, but it's also kind of bad, because that means when you're debugging your program, you can get very confused. You say, it says it's one, why am I getting a different answer when I do the computation? And that's because it's not really one inside. So you have to be careful.

Now mostly, these round-off errors balance each other out. Some floats are slightly higher than they're supposed to be, some are slightly lower, and in most computations it all comes out in the wash and you get the right answer. Truth be told, most of the time, you can avoid worrying about these things. But, as we say in Latin, caveat computor. Sometimes you have to worry a little bit.

Now there is one thing about floating points about which you should always worry. And that's really the point I want to drive home, and that's about the meaning of double equal.

Let's look at an example of this. So we've before seen the use of import, so I'm going to import math, it gives me some useful mathematical functions, then I'm going to set the variable a to the square root of two.

Whoops. Why didn't this work? Because what I should have said is math dot square root of two. Explaining to the interpreter that I want to get the function sqrt from the module math. So now I've got a here, and I can look at what a is, yeah, some approximation to the square root about of two.

Now here's the interesting question. Suppose I ask about the Boolean a times a equals equals two. Now in my heart, I think, if I've taken the square root of number and then I've multiplied it by itself, I could get the original number back. After all, that's the meaning of square root.

But by now, you won't be surprised if the answer of this is false, because we know what we've stored is only an

approximation to the square root. And that's kind of interesting. So we can see that, by, if I look at a times a, I'll get two point a whole bunch of zeros and then a four at the end.

So this means, if I've got a test in my program, in some sense it will give me the unexpected answer false. What this tells us, is that it's very risky to ever use the built-in double--equals to compare floating points, and in fact, you should never be testing for equality, you should always be testing for close enough.

So typically, what you want to do in your program, is ask the following question: is the absolute value of a times a minus 2.0 less than epsilon? If we could easily type Greek, we'd have written it that way, but we can't.

So that's some small value chosen to be appropriate for the application. Saying, if these two things are within epsilon of each other, then I'm going to treat them as equal.

And so what I typically do when I'm writing a Python code that's going to deal with floating point numbers, and I do this from time to time, is I introduce a function called almost equal, or near, or pick your favorite word, that does this for me. And wherever I would normally written double x equals y, instead I write, near x,y, and it computes it for me.

Not a big deal, but keep this in mind, or as soon as you start dealing with numbers, you will get very frustrated in trying to understand what your program does.

OK. Enough of numbers for a while, I'm sure some of you will find this a relief. I now want to get away from details of floating point, and talk about general methods again, returning to the real theme of the course of solving problems using computers.

Last week, we looked at the rather silly problem of finding the square root of a perfect square. Well, that's not usually what you need. Let's think about the more useful problem of finding the square root of a real number.

Well, you've just seen how you do that. You import math and you call sqrt. Let's pretend that we didn't know that trick, or let's pretend it's your job to introduce-- implement, rather-- math. And so, you need to figure out how to implement square root.

Why might this be a challenge? What are some of the issues?

And there are several. One is, what we've just seen might not be an exact answer. For example, the square root of two. So we need to worry about that, and clearly the way we're going to solve that, as we'll see, is using a concept similar to epsilon. In fact, we'll even call it epsilon.

Another problem with the method we looked at last time is, there we were doing exhaustive enumeration. We

were enumerating all the possible answers, checking each one, and if it was good, stopping.

Well, the problem with reals, as opposed to integers, is we can't enumerate all guesses. And that's because the reals are uncountable. If I ask you to enumerate the positive integers, you'll say one, two, three, four, five. If I ask you to enumerate the reals, the positive reals, where do you start? One over a billion, plus who knows?

Now as we've just seen in fact, since there's a limit to the precision floating point, technically you can enumerate all the floating point numbers. And I say technically, because if you tried to do that, your computation would not terminate any time soon.

So even though in some, in principle you could enumerate them, in fact you really can't. And so we think of the floating points, like the reals, as being innumerable. Or not innumerable, as to say as being uncountable. So we can't do that.

So we have to find something clever, because we're now searching a very large space of possible answers. What would, technically you might call a large state space. So we're going to take our previous method of guess and check, and replace it by something called guess, check, and improve.

Previously, we just generated guesses in some systematic way, but without knowing that we were getting closer to the answer. Think of the original barnyard problem with the chickens and the heads and the legs, we just enumerated possibilities, but we didn't know that one guess was better than the previous guess.

Now, we're going to find a way to do the enumeration where we have good reason to believe, at least with high probability, that each guess is better than the previous guess.

This is what's called successive approximation. And that's a very important concept. Many problems are solved computationally using successive approximation.

Every successive approximation method has the same rough structure. You start with some guess, which would be the initial guess, you then iterate-- and in a minute I'll tell you why I'm doing it this particular way, over some range. I've chosen one hundred, but doesn't have to be one hundred, just some number there-- if f of x, that is to say some some function of my--

Whoops, I shouldn't have said x. My notes say x, but it's the wrong thing-- if f of x, f of the guess, is close enough, so for example, if when I square guess, I get close enough to the number who's root I'm-- square root I'm looking for, then I'll return the guess.

If it's not close enough, I'll get a better guess. If I do my, in this case, one hundred iterations, and I've not get--

gotten a guess that's good enough, I'm going to quit with some error. Saying, wow. I thought my method was good enough that a hundred guesses should've gotten me there. If it didn't, I may be wrong.

I always like to have some limit, so that my program can't spin off into the ether, guessing forever.

OK. Let's look at an example of that. So here's a successive approximation to the square root. I've called it square root bi. The bi is not a reference to the sexual preferences of the function, but a reference to the fact that this is an example of what's called a bi-section method.

The basic idea behind any bi-section method is the same, and we'll see lots of examples of this semester, is that you have some linearly-arranged space of possible answers. And it has the property that if I take a guess somewhere, let's say there, I guess that's the answer to the question, if it turns out that's not the answer, I can easily determine whether the answer lies to the left or the right of the guess.

So if I guess that 89.12 is the square root of a number, and it turns out not to be the square root of the number, I have a way of saying, is 89.12 too big or too small. If it was too big, then I know I'd better guess some number over here. It was too small, then I'd better guess some number over here.

Why do I call it bi-section? Because I'm dividing it in half, and in general as we'll see, when I know what my space of answers is, I always, as my next guess, choose something half-way along that line. So I made a guess, and let's say was too small, and I know the answer is between here and here, this was too small, I now know that the answer is between here and here, so my next guess will be in the middle.

The beauty of always guessing in the middle is, at each guess, if it's wrong, I get to throw out half of the state space. So I know how long it's going to take me to search the possibilities in some sense, because I'm getting logarithmically progressed.

This is exactly what we saw when we looked at recursion in some sense, where we solved the problem by, at each step, solving a smaller problem. The same problem, but on a smaller solution space.

Now as it happens, I'm not using recursion in this implementation we have up on the screen, I'm doing it iteratively but the idea is the same. So we'll take a quick look at it now, then we'll quit and we'll come back to in the next lecture a little more thoroughly.

I'm going to warn you right now, that there's a bug in this code, and in the next lecture, we'll see if we can discover what that is.

So, it takes two arguments; x, the number whose square root we're looking for, and epsilon, how close we need to

get. It assumes that x is non-negative, and that epsilon is greater than zero.

Why do we need to assume that's epsilon is greater than zero? Well, if you made epsilon zero, and then say, we're looking for the square root of two, we know we'll never get an answer. So, we want it to be positive, and then it returns y such that y times y is within epsilon of x. It's near, to use the terminology we used before.

The next thing we see in the program, is two assert statements. This is because I never trust the people who call my functions to do the right thing. Even though I said I'm going to assume certain things about x and epsilon, I'm actually going to test it. And so, I'm going to assert that x is greater than or equal to zero, and that epsilon is greater than zero.

What assert does, is it tests the predicate, say x greater than or equal to zero, if it's true, it does nothing, just goes on to the next statement. But if it's false, it prints a message, the string, which is my second argument here, and then the program just stops.

So rather than my function going off and doing something bizarre, for example running forever, it just stops with a message saying, you called me with arguments you shouldn't have called me with.

All right, so that's the specification and then my check of the assumptions. The next thing it does, is it looks for a range such that I believe I am assured that my answer lies between the ran-- these values, and I'm going to say, well, my answer will be no smaller than zero, and no bigger than x. Now, is this the tightest possible range? Maybe not, but I'm not too fussy about that. I'm just trying to make sure that I cover the space.

Then I'll start with a guess, and again I'm not going to worry too much about the guess, I'm going to take low plus high and divide by two, that is to say, choose something in the middle of this space, and then essentially do what we've got here.

So it's a little bit more involved here, I'm going to set my counter to one, just to keep checking, then say, while the absolute value of the guess squared minus x is greater than epsilon, that is to say, why my guess is not yet good enough, and the counter is not greater than a hundred, I'll get the next guess.

Notice by the way, I have a print statement here which I've commented out, but I sort of figured that my program would not work correctly the first time, and so, I, when I first typed and put in a print statement, it would let me see what was happening each iteration through this loop, so that if it didn't work, I could get a sense of why not.

In the next lecture, when we look for the bug in this program, you will see me uncomment out that print statement, but for now, we go to the next thing.

And we're here, we know the guess wasn't good enough, so I now say, if the guess squared was less than x, then I will change the low bound to be the guess. Otherwise, I'll change the high bound to be the guess. So I move either the low bound or I move the high bound, either way I'm cutting the search space in half each step. I'll get my new guess. I'll increment my counter, and off I go.

In the happy event that eventually I get a good enough guess, you'll see a-- I'll exit the loop. When I exit the loop, I checked, did I exit it because I exceeded the counter, I didn't have a good-enough guess. If so, I'll print the message iteration count exceeded. Otherwise, I'll print the result and return it.

Now again, if I were writing a square root function to be used in another program, I probably wouldn't bother printing the result and the number of iterations and all of that, but again, I'm doing that here for, because we want to see what it's doing.

All right. We'll run it a couple times and then I'll let you out for the day. Let's go do this. All right. We're here.

Well, notice when I run it, nothing happens. Why did nothing happen? Well, nothing happens, it was just a function. Functions don't do anything until I call them. So let's call it. Let's call square root bi with 40.001 Took only one at-- iteration, that was pretty fast, estimated two as an answer, we're pretty happy with that answer. Let's try another example. Let's look at nine. I always like to, by the way, start with questions whose answer I know. We'll try and get a little bit more precise.

Well, all right. Here it took eighteen iterations. Didn't actually give me the answer three, which we know happens to be the answer, but it gave me something that was within epsilon of three, so it meets the specification, so I should be perfectly happy.

Let's look at another example. Try a bigger number here.

All right? So I've looked for the square root of a thousand, here it took twenty-nine iterations, we're kind of creeping up there, gave me an estimate.

Ah, let's look at our infamous example of two, see what it does here. Worked around.

Now, we can see it's actually working, and I'm getting answers that we believe are good-enough answers, but we also see that the speed of what we talk about as convergence-- how many iterations it takes, the number of iterations-- is variable, and it seems to be related to at least two things, and we'll see more about this in the next lecture. The size of the number whose square root we're looking for, and the precision to which I want the answer.

Next lecture, we'll look at a, what's wrong with this one, and I would ask you to between now and the next lecture,

think about it, see if you can find the bug yourself, we'll look first for the bug, and then after that, we'll look at a better method of finding the answer.

Thank you.