The following 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: OK, I want to remind you that there's a quiz one week from today. Yeah, I know it's soon. Open book, open notes, no computing or communication devices allowed. Between now and then, probably tomorrow in fact, or at least over the weekend, I'll send out a summary of what I think we've covered so far and what you'll be responsible for in the quiz. Roughly speaking, it's anything covered in lectures, problem sets, or recitations. I will also post some practice questions that you can work on. And I'll tell you now that we will not be posting answers to the practice questions. Instead, we'll be holding some quiz reviews. OK.

I wanted to cover two different topics today. The first topic is just a tiny bit on floating point numbers in Python. But in fact, what I'm going to tell you is true about all programming languages-- in fact all, computers really. And then after that we'll spend most of the lecture on the topic of debugging.

So let me start with a quick review of binary numbers. Because you have to understand binary numbers to understand floating point. So when you first learned about numbers, you learned about base 10. And you learned that a decimal number is represented by some combination of the digits 0 through 9, the rightmost place is the 10 to the 0 place, and then it's the 10 to the 1 place, the 10 to the 2 place, et cetera. So for example, the number 302 or the digits 3-0-2 represent 3 times 100, plus 0 times 10, plus 2 times 1. Duh.

All right, binary numbers are exactly the same except we only have two digits to choose from. Typically written as 0 and 1 and everything is represented by a sequence of those digits. The rightmost place is 2 to the 0, the next place is 2 to the 1, 2 to the 3, 2 to the 4, et cetera. So for example, if we look at the binary number 1-0-1, we see that's equal to 1 times 4, plus 0 times 2, plus 1 times 1, or 5.

So one of the first things we'll notice is binary numbers take a lot more digits to represent them, or take more digits than decimal numbers. In fact, if I give you n digits, n binary digits, how many different binary numbers can I represent with those n digits?

Well, if I gave you n decimal digits, how many different numbers can I represent? How many different values can I represent?

AUDIENCE: 10 to the n.

PROFESSOR: Pardon?

AUDIENCE: 10 to the n.

PROFESSOR: 10 to the n. And so, for a binary number it's going to be 2 to the n.

That's important, because we'll see as we get to talking about the complexity of various algorithms how long they take to run, or how much space they use, we'll frequently be resorting to arguments of this sort to understand them.

Now the reason floating point numbers cause problems for programmers is that people have learned to think in base 10. Computers do everything in base 2, and that causes a cognitive dissonance sometimes. Where people are thinking one thing, and the computer is doing something slightly different.

So why do people work in base 10? I don't know. Maybe it's because we have 10 fingers, but we also have 10 toes. So why didn't we work in base 20? We have one head, I don't know why. But we do it, we work in base 10. I do know why computers work in base 2. And that's because it's easy to build switches in electronic hardware.

A switch is some physical device that has only two possible positions, on or off. We can build very efficient switches in hardware and so it's easy to represent a number as a sequence of on and off bits, which is either on or off. Originally they were relays, then they became transistors, now they're something altogether different. But, what they all had in common was they were stable in the off position, they were

stable in the on position, and they never had to get in between. Hence, we represent everything in computers in binary.

So now let's think about why that causes some confusion. And it does only for fractional numbers. So for whole numbers binary and decimal it doesn't matter. Ints are never confusing, they sort of do what God told us integers should do, or whoever told us integers.

All right, but now let's look at other things. So I want to start by looking at the decimal number 0.125. What's that as a fraction, by the way? Happens to be one what?

AUDIENCE: 1/8.

PROFESSOR: 1/8, we'll see why that actually matters in a minute. So, what does it mean, in some sense, in decimal? It's equal to 1 times 10 to the minus 1 plus 2 times 10 to the minus 2 plus 5 times 10 to the minus 3. So it works exactly the same way that things work on the other side of, in this case, the decimal point.

Suppose we want to represent it in binary. So instead of a decimal point, we have a binary point. What does it look like then? Well it's equal to what? 1 times-- if it's 1/8, what's it going to be? 1 times what?

AUDIENCE: 1 times 10 to the minus 3.

PROFESSOR: 10 to the minus 3. Or, 0.001. Right? So, so far, so good. Not much difference between the two.

Now let's take a different decimal number. What about the decimal 0.1? I have to tell you that it's a decimal because it could also be a binary with just 0's and 1's. Well, we know how to represent that in decimal. How about in binary? What's the equivalent? Now that's 1/10, of course. What does 1/10 look like in binary? Any takers? Well I'll give you a hint. It's so long, that I don't want to write it on the board. In fact, it's worse than long, it's infinite. I guess that's kind of long. It's this repeating binary fraction. There is no finite combination of binary digits that represent the

decimal fraction 1/10. There's no way to do it.

And that's why things get a little hairy. So we can stop at some finite number of bits. And in fact that's what happens in the internal representation in Python. It ends up representing 1/10 as something equivalent to this decimal fraction. If I take the number of binary bits that are inside the computer, and then I translate it back to decimal, it turns out that it's using this approximation for the decimal fraction 1/10.

So for example, some of you in your problem sets -- where you were computing how much you had to pay on a credit card -- would get answers that were eventually off by a penny or something from what we expected in some, and that has to do with the fact that you were thinking in decimal. And in fact, you were writing your program in decimal, yet internally things were happening in binary, and when you thought you were writing 1/10 for example you were actually getting something like this inside the computer. Pretty close to 1/10, but not exactly 1/10.

Now, when we print it, we get yet something else because the print statement uses an internal function that by default rounds these things to 17 digits. And so you end up getting something like that, or you might depending how you do it. So let's look at an example here.

So I can do something like this, and it prints that because it's doing some rounding for me. But if I really look at what's under there, and look at the representation, the REPR function is convenient to get a sense of what's really going on inside, it tells me that well that's a 17-digit approximation. And now so that's what's really lurking there. So a hint, If you think something is going funny because of the way arithmetic is working, instead of just using print, you can use print of REPR to get a better idea about what's really going on.

All right, now, does this matter? Usually it doesn't. Most of the time it's safe just to pretend that floating points work the way you learned about arithmetic when you were in third grade, or probably in kindergarten if you were educated in Europe or Asia.

But now let's look at an example where you can get in trouble. So I've got a little program here. I initialize x to 0, then I'm going to go through a loop a lot of times, where I increment x by 1/10. And then I'm going to print x. And because it's going to do automatic rounding, it's going to print 10,000-- or actually, it should print 100,000, right? No 10,000, because I'm only incrementing it by 1/10, excuse me. But then I'm going to print REPR of x, and then I'm going to do a comparison.

Now if floating point arithmetic worked the way reals work, we would think that 10.0 times x should equal the number of iterations. Because I'm starting at 0, each time I'm incrementing it by 1/10, and so if I multiply the result by 10 at the end, I should get the same as the number of iterations. Does that make sense to everybody? That's what you would normally get if you did this with pencil and paper. Of course, it would take you a really long time to do 100,000 increments. Let's give it a shot. And what we'll see is that if I print it, it looks OK, it's 1,000. But if I print REPR of it, I see it's 10,000, a bunch of 0's, and then 18848. And, of course, consequently when I compare it, I get something that says false. And that's because if I look at REPR of 10.0 times x-- well, that's interesting, what's going on here? It kind of looks like the same thing, doesn't it? But it's not, because way out there are some other digits we're not seeing, something different is happening.

OK, what's the moral of this? It's not complicated. It's not, OK write your programs thinking deeply about what's going on in those bits way out there at the end. It's, don't ever test whether to floating numbers are equal to each other. Instead, do something like this. Define a function called 'close', or whatever you want, that takes two floats and some epsilon. And I've given here epsilon a default value. And then just return whether the absolute value of x minus y is less than epsilon. So whenever you're comparing two floating numbers, the question shouldn't be are they identical, but are they close enough for your purposes. And if you do that, then you don't get tripped up by this kind of rounding and things like that.

Not a complicated story, but keeping this in mind will get you out of trouble when you're doing floating point arithmetic.

Let's run this, and see what happens. And indeed, they're not equal but they're good enough, close enough if you will. OK.

One of the dangers, the reason this went wrong, is these little differences can accumulate if you go through a lot of iterations. Sometimes they balance out, sometimes it rounds up, sometimes it rounds down, but not always. So very simple answer. Just don't get caught up in this problem of floating point numbers.

All right, any questions about that? All right, Yes.

AUDIENCE: Doesn't it change for Python 2.7? It's only returning 0.1 and not 0.100000.

PROFESSOR: In 2.7?

AUDIENCE: Yeah.

PROFESSOR: Don't know, sorry. But the moral remains the same. Whatever is going on, don't test floating point numbers for quality because you'll have a high probability of getting false, when you should get true. OK. You almost never get true when you should get false.

I now want to move on if there are no more questions to debugging. I never know when to give this lecture in the term. So what I usually do is I wait until the volume of email, and complaints, and office hours builds, and I realized people are ready to learn more about debugging. If I do it too early, people don't pay any attention because they don't realize it's a problem. And if I do it too late, they get irritated with me because they say well why didn't you tell me this earlier in the semester when it would've done me some good. So, I pick a time. And right now it looks like the need has built up enough that it's worth doing.

There's a very charming urban legend about how the process of fixing flaws in software came to be known as debugging. It's one of those stories that's so nice that you just want it to be true. So let's look at this story, because it's fun.

All right, what you see on the screen now is a photo of a book now at the Smithsonian Museum, of the lab book from the group working on the Mark II Aiken

Relay computer at Harvard University. Pardon? Oh, I see it on my screen, now you see it on your screen. Thank you.

So there it is. It was September 9, 1947, even before I was born, it was that long ago. And so you can see that they're running their computer, and they started to do an arctan computation, and it's kind of interesting that they started it at 8 o'clock in the morning, and it ran for two hours, and then it stopped. Wow, to do an arctan. Tells you something about how fast this computer was. Then it went on. Then they started the cosine tape, and started to do a multiple adder, and then something bad happened. It stopped working. Whoops. All right, hold on a second.

And they spent a long time trying to find out why it stopped working. And then they found out the problem. They found a moth stuck between one of the relays. So it had electromechanical relays for their switches, the on and off. And they were debugging, they didn't call it debugging. And they found the software had failed because the hardware had failed, and the hardware had failed because a bug had been stuck in one of the relays. They debugged it, as in removed the moth, and the program ran to successful completion.

And as you can see, the comment was written in this book, first actual case of a bug being found. Hence, we call it debugging. This was, by the way, Grace Murray Hopper's lab book. She is often described as the first programmer. It's unclear if that's true. What is true, she was the first female Admiral in the US Navy. She was a Navy programmer who eventually rose to the rank of Admiral.

So it's a charming story that this is why we call it debugging. Turns out it's not at all true. That the phrase debugging had been used for a long time, and could easily be traced back to the 1800s when people were writing books about electronics and talking about debugging even in those days. And in fact, you can go back to Shakespeare who talks about a bugbear, meaning something causing needless exercise, needless or excessive fear or anxiety. Well that's a good description of a bug. And he actually called it a bug when he had Hamlet [UNINTELLIGIBLE] about to bugs and goblins in my life.

All right, so I want to start now-- oh by the way, just for fun, this is what the Mark II looked like. This was the computer the took an hour or so to do an arctan. You see it filled-- made it's a little hard to see in this light-- but you can see it filled an entire room. Quite amazing. And, here's a picture of Admiral Hopper and some unidentified mail. All right, if anyone knows who this it would be good to know so I can update my archives.

All right, so now on to some practical aspects of debugging. The first thing I want to do is dispel some myths about debugging. There is this myth that bugs crawl unbidden into our programs. That we write perfect programs and somehow a bug just sneaks in, and ruins perfection. That's not true. In fact, if there's a bug in your program, you put it there. So it would be almost better not to call it a bug, which sort of sounds like it's not our fault, but it's a mistake, it's a screw up. So get that through your head.

Similarly bugs do not breed in programs. If there are multiple bugs in your program, it's not because a couple of them got together and procreated, it's because you made a lot of mistakes. Keep that in mind. With that in mind, we should think about what the goal of debugging-- and it's not to eliminate one bug quickly, it is to move towards a bug-free program. And I say this because they're not always the same strategy that you would follow for these different goals. And I also carefully say to move towards a bug-free program because in truth be told we are hardly ever sure that we have no bugs left.

Debugging is a learned skill. Don't despair. Nobody does it well instinctively. Evolution did not train us to be debuggers. So a large part, probably the largest part in many ways, of learning to be a good programmer is learning to debug. And what that has to do is thinking systematically and efficiently about how to move towards a bug-free program.

The good news is that it's not hard to learn, and it is a largely transferable skill. The same skills you use to debug software, can be used to debug laboratory experiments. I actually give lectures sometimes to physicians about how to debug

patients. How to use debugging techniques to find out what's wrong with people when they're sick. It's a very good and useful life skill.

Now for four decades, maybe five decades, people have been building tools called the debuggers. And you'll find that built into IDOL there is a debugger that are designed to help people find out why their programs don't work, and fix them. Personally, I almost never use one. The tools are not that important. What's important is the skill of the craftsman, in this case. And in fact, most of the experienced programmers I know rely on print statements.

So it's OK to use a debugger but I think the best debugging tool is print. And I have to say I've been surprised-- that's a mild word here-- at how few print statements you guys seem to use. I get these emails, or the staff gets these emails, kind of plaintiff, why doesn't my program work? And then there's a little piece of code. And the answer I send back-- when I reply before one of the TA's do, and they usually get there first-- is usually, put in a print statement here and see what happens.

And I'm just amazed that when the code arrives it doesn't have these statements in it. My favorite response, was I sent an email to a student, who shall go nameless, and he-- or maybe it was a she-- and I said, insert a print statement here and see what happens. And I got back to reply saying, no I don't need a print statement here I know what the value of this variable is. Well, you know, my reply was that if all the values were what you thought they were, you wouldn't be sending an email saying, why doesn't my program work. Put the darn print statement and see what happens. And then I got a gracious email back saying, more or less, oops, I see.

But please, when you send us some code, you want some help, send us code with some print statements already in it to at least show us that you've tried to find the bug yourself.

All right, so what we're essentially doing when we insert print statements in a code is searching for the place in our program where things have gone awry. And the key to being a good debugger is to be systematic in this search. So you saw that when we looked at algorithms for things like exhaustive enumeration. We said, well if we're

searching for an answer, we have to search the space carefully one at a time. And then we said, if we want to search it efficiently, maybe instead of starting at the beginning and just going to the end, we should use something like binary search.

The same techniques can be used when you're searching for bugs. So I recommend searching for bugs using some approximation to binary search. And we'll see an example of this as we go forward, but as we look at the example what I want you to think about is what are we searching for? We know our program doesn't work. So the question that I like to ask, is not why didn't it produce the answer I wanted it to? But, how could it have done what it had done?

This is a subtly different question. And it's usually a much easier question to answer. Not why didn't it do the right thing, but here it did something. So I already know what it did. And I say, I didn't expect it to do that, so why did it do that? Once I know why it did what it did, it's usually pretty easy to think how to fix it. So that's the first question I ask.

I then go about it using something akin to the scientific method, which we all learned about many years ago. And basically the scientific method is based upon studying available data. The data you have is of course the program text itself, the test results, you ran some tests and got the wrong answer which is why you knew you had a bug. And then you can probe it, you can change the test results by using print statements so that you have more data to study. Keep in mind that you don't understand this program, because if you did it would work.

Once I study this, I form a hypothesis that at least I think is consistent with the data. And then I go and design and run a repeatable experiment. And I want to emphasize the word repeatable, here. And again the key thing as with the scientific method, the experiment to be useful must have the potential to refute the hypothesis.

Why might repeatability to be an issue? Well, as we'll see pretty soon, a lot of programs involve randomness. Where you're doing something equivalent to flipping a coin, somewhere in the program which might come up heads or tails, and the

program would do different things. We'll see why that's an important programming techniques soon. And once you do that, you can get different results with different runs.

More subtly there can be various kinds of timing errors deep down in the operating system where you have multiple activities going on at the same time. This is usually the reason that you'll see say, Windows crash, or Word, or PowerPoint, or something else. Because there's some timing error that occurs sometimes. And probably most commonly, because there's human input. Somebody typed something and they might type something different.

So one of the things you want to do when you're systematic is make sure that you can replay things. And we'll talk more about this when we get to randomness, about how we go about doing that.

All right, now let's try and put this all together in a little program. If you've been studying your handout, as at least one of the TA's did, you've been kind of mystified by the fact that there's a pretty crummy looking program in it. And unlike sometimes when I make mistakes I don't know I've made, here I intentionally made some mistakes. So let's look at this program. I wrote a function called is_palindrome that takes in a list and is intended to return true if the list is a palindrome and false otherwise. Then I wrote this little program called Silly that uses isPal, takes in a number, requests that the user make that many inputs, then calls isPal to find out whether or not the resultant list is a palindrome. Not too complicated. But now let's run it.

Do Silly of 'five'. And it tells me 'abcde' is a palindrome. All right, I have a bug. Now I need to go try and find that bug. So the first thing I need to think about when I'm looking for it is to try and find a smaller piece of input that will produce the bug.

So I want to find small input on which program fails. Why do I want to find a smaller input? Well, a in this case it's less typing, b if it's a real program it's probably less execution time to make it run, but c it'll be easier to debug because there are fewer kinds of problems. So let me try it on a small piece of input say, Silly of 1. Oh, it gets

that right. So that's no good. Let me try something else, let's try Silly of 2, I'm sort of sneaking up. It gets that one wrong. All right, so I know I can test it on a small input. So that's a good thing. I now have a simple test.

Now in this case the code is so short, and so stupid, that you could probably look at it with your eyes and just find the bug instantly. But the point of this exercise is not to find the bug, but to kind of show the process.

So now I wanted to go through this process of binary search to try and find the bug. So we'll start with Silly, the top level program, and I'll look for something about halfway through, maybe here. And try and now answer the question, that I've got a lot of code and I'm going to find a point halfway through it and try and ask is the bug above this, or below this. So I need to find some intermediate value I can check. And at this point in the program the only thing I have done is accumulate the input, right? So there's nothing else to ask. So my hypothesis is that everything is good and that the input will be 'ab'. So let's try it. Let's print result here every time through and see if we get what we wanted to get.

All right, that's not what I expected. So something is wrong. What's wrong? Why is result always the empty list? I can out-wait you.

AUDIENCE: Because whenever it goes through the for loop it keeps coming back.

PROFESSOR: Right. So every time through the for loop, it's reinitializing-- whoa, got you. For those of you watching on TV, I just hit a person that was heads down with a piece of candy. Fortunately it was not a hard candy. All right, so you're right. Let's get that out of there. Put it where it belongs. Run it again. OK, are we happy with that result? Yeah, because I've done that before the append, right? And now just to be sure, we'll take this print statement out here and let's put it here. We're now searching elsewhere.

Well the good news is I now have the right result for the value of the variable, but the wrong result for the program. It's still telling me it's a palindrome. So the moral here is there is no such thing as the bug. Never use the definitive article. There is a

bug.

There's a story that I've heard related to this, as far as finding a bug. You can imagine that you're at someone's house for dinner, you're sitting at the dining room table, you can't see the kitchen, and suddenly you hear from the kitchen, [BAM]. What the heck's that? Your hostess walks out and says, don't worry I just killed the cockroach on the turkey. Well, your immediate reaction is the cockroach on the turkey? Where there's one, there's likely to be more.

Every time you found a bug-- the more bugs you find, then probably the more bugs there are still left, because you've shown that you make a lot of mistakes.

All right, onward we go. So what do we do next? Well, we now know at least that things look OK to this point, which suggests that the problem must come below this in the program. Well the only thing that's going on below this is the call to isPal. So now we'll say OK, we've now isolated the bug to isPal. That's a good thing. Let's try and ask where things are going on there.

So we'll take a point halfway through isPal, and we'll print some things here. So let's print-- see what we have here. But before I do that, I've gotten really tired of typing 'a' and 'b', so I'm going to use something called a test driver, or a test harness. And I recommend that you do this kind of thing whenever you're testing a program. Write some code that has nothing to do with the program itself but makes it easier to test and debug the program.

The pretentious word for this is a test harness. All this is code that helps testing. One of the things that you see in industry is about half the code that gets written is not intended to be delivered as part of the final product, but is there merely for the purpose of testing and debugging. It's a big deal. So don't feel bad that you're writing code that's not part of the solution to the problem set that is there only to help you make your code work.

It seems like it's extra work, but in fact, it will save you work. So let's call it. We'll call isPal. And it's going to print some things that I think it should do. In fact, we'll look at

what it does first before we look at the print statements in isPal. So for the moment, let me just comment these out.

And what we see here is it should print false, and it prints true. Well, should it print false the second time? No, right. So it should have printed true, and it did. So this is an important lesson. Make sure that when you put in these debugging statements, you write down as part of the print statement what you expect it to print. So that when you look at your output you can quickly scan it and see whether the program is behaving as you thought it would.

So now, works once doesn't work the other time. So we'll go back and turn on the print statements up here and see what we get. So it's printed temp as 1-2-1 and x as 1-2-1. So kind of OK that print and x are the same, we expected that. But we thought we reversed it. We've entered 1-2-1 and it is this. What's going on? What's wrong?

Well now what we can do, is let's see where it went wrong. We'll put in another print statement here, see what value is there. Well it was 1-2-1 before reverse, and it's 1-2-1 after reverse. How come? Why isn't reverse reversing temp?

AUDIENCE: Do you need parenthesis after reverse?

PROFESSOR: Exactly, I need parenthesis after reverse. Whoa, close. Because without the parentheses, all reverse is doing is nothing. That's just the name of the method, not an invocation of the method, right?

All right, now let's run it. Good news and bad news. What's the good news? It has indeed reversed 1-2 right, to make it 2-1 but it's also reversed x. So naturally, since it's reversed x temp and x will be the same, and I get the wrong answer. What's wrong now? Yeah?

AUDIENCE: So, I think you're aliasing.

PROFESSOR: I'm aliasing?

AUDIENCE: And it's reversing--

PROFESSOR: Because now remember how mutation works, now temp and x both point to the same object. If I reverse the object, it doesn't matter whether I get to it through x or I get to through temp it will still have been reversed. So in this case, what I'd need to do is this, clone it. And now when I run my code, it works. No applause? All right, a couple more things about debugging next Tuesday, and then we'll move on to some pretty interesting topics in the next phase of the course.