PROFESSOR: Good morning. This is 600. So I hope any of you who thought this was a different course find where you really belong. My name is John Guttag. And I'll be lecturing the course all semester.

OK. Today I want to accomplish several things. Cover some of the administrative details. Talk about the goals of the course-- what I hope you'll learn. And then begin talking about the conceptual material in the course.

It will seem a little bit slow, because it will be a little bit slow. I promise starting on Thursday we're going to pick up the pace considerably. So let's start with the strategic goals of the course.

The official introduction to course 6 is 601. Historically, students who arrive at MIT with little or no programming experience find 601 an ordeal. And the point of this is to prepare freshman and sophomores for entering course 6-- that's the Electrical Engineering Computer Science department-- in a gentler, kinder way. So that 601 is not so much of a problem.

I want to help students feel justifiably-- and I want to emphasize the word justifiably-- confident in their ability to write small and medium sized programs. So at the end of the term, you should all feel comfortable writing programs.

The real theme of the course, and what I spend most of the time on, is how to map problems into a computational framework. There's going to be an emphasis on scientific problems, rather than, say, commercial problems.

But there will be some talk about some non-scientific problems as well. How to take a problem that may not at first blush appear to be attackable with the program, and

show you how to formulate the problem in such a way that you can use computation to get insight into the problem.

It should not take you very long. All of the problem sets involve programming in Python programming language, which I'll say a little bit about later today. The first problem set, basically, is getting Python installed on your own computer. Most of the people will want to just use whatever their own laptop is to do the problem sets on.

Don't get fooled by the first two problem sets into thinking this is a gut course. It's not. It starts out gently to lure you in. And then life gets pretty hard pretty quickly. So don't be fooled.

The quizzes-- and there will be two evening quizzes and a final-- are open book open notes. When you get to be my age, you get very sensitive about how difficult it is to remember things. And so, we won't be asking you to memorize stuff. The course is about solving problems, knowing how to solve problems, not how much you can remember.

For many of you who are majoring in biology or course 20, it's going to be kind of a shock that this course isn't about how much you can remember, but it's about how well you can solve problems. And that's what the quizzes are really going to be focused on.

Probably the most unusual thing about this course is the collaboration policy, which is liberal in the extreme. You can collaborate with anybody you want on any of the problem sets. Not on the quizzes. But on any of the problem sets.

You can work with each other, which is what I recommend. You can work with your parents, if one of them happens to be a software engineer. You can work with friends in course 6. Whatever you want to do.

The goal of the problem sets is to help you learn. What we've seen in the past is people who are a little, shall I say too collaborative, i.e. they just copy the problem set from somebody else, live in a fool's paradise which comes crashing down at the first quiz. People who don't spend enough time thinking about the problem sets

themselves cannot take the quizzes successfully.

So it's a fine line. But our goal is not to be policemen. I tell my TAs, their job is to help you learn, not to prevent you from, quote, cheating. So to solve that problem, we've eliminated the concept of cheating on problem sets. There is no way to cheat on problem sets. So just go and do them.

There's no textbook. We will be posting readings on the web. For the most part, these will be pointers to websites. Occasionally we'll post readings that we wrote ourselves. Doesn't mean you shouldn't buy a textbook.

In fact, there are a number of Python texts. We'll recommend a few of them. It might make sense to buy one and bring it to a quiz, because it will have an index that will let you look things up quickly, that sort of thing. But again, a lot of students never buy a text and do just fine.

We will not be handing out class notes on a regular basis. A lot of studies have indicated that students learn more when they take their own notes than when they are handed out. And so, as a matter of what I think is good pedagogy, we don't hand out detailed lecture notes.

We will be using, after today, a lot of handouts with code on them, which we'll make available. But it's not intended to make any sense outside the context of lectures. It's not self-contained.

The main purpose of this course is to help you become skillful in making the computer do what you want it to do. Once you acquire this skill, your first instinct, when confronted with many tasks, will be to write a program to do that task for you.

I always tell people I became a computer scientist in part because I'm lazy. And there was a lot of stuff that I found it was easier to write a program to make the computer do it, rather than do it myself. So I do that a lot. If I need to do something, I say, can I just write a quick program to do it? And I'd like you to be able to acquire that skill.

And remember, that programming is actually a lot of fun. So I should say that, in addition to learning a lot in this course, I hope most of you will find it fun. Kind of a strange thought. MIT course, fun. Maybe it's an oxymoron. But I don't think so.

I really do think you can have a lot of fun writing programs for this course. There are many people who believe that-- how shall I say this? Programming is the most fun you can have with your clothes on. It really can be a lot of fun. So think of it that way.

All right. So the primary knowledge you're going to take away is computational problem solving. So to start with, we might ask the question, what is computation? And to think about that-- this is interesting.

Ah, there's where the chalk is. I was afraid I was going to be confronted with a sea of black boards and erasers and no chalk. But there is chalk.

So if we think about it, there are essentially two kinds of knowledge. Declarative-- and you're going to see, I'm not a great speller. And imperative.

Declarative knowledge is composed of statements of fact. For example, a good health care plan improves the quality of medical care while saving money. As we know from doings in Washington, it's a lot easier to state that goal than to know how to achieve that goal.

So the key thing about declarative knowledge is, it says something that is true. Otherwise it wouldn't be knowledge, it would be misinformation. But doesn't tell you how to do it. In a more mathematical sense, say y is the square root of x if and only if y times y equals x.

All right? Perfectly clear statement of what it means to be the square root, but it doesn't tell you how to find the square root. Interestingly enough, it does tell you how to test whether or not you have the answer to the square root. And so if you had some way of generating guesses, you can at least check whether they're correct.

And in fact, starting in the next lecture, we'll talk about the fact that a lot of computational techniques involve something called guess and check. Where you have a way to generate guesses and a way to check whether they're right.

Imperative knowledge, in contrast, tells you how to solve a problem. How to accomplish something. So you could think of it as like a recipe in a cookbook. So it's one thing to say a chocolate cake is something that tastes delicious and is bad for you. That's declarative knowledge. But you can open a cookbook and get a recipe that tells you how to make a chocolate cake. That's imperative knowledge.

Now, below, we have a recipe for finding not a square root necessarily, but an approximation to a square root. And one of the themes of this course is that a lot of problems we cannot solve precisely, but we can find answers that are good enough for practical purposes. And those are called approximation algorithms.

So here's a way-- this is a very old method for finding the square root. In fact, it's believed that Heron was the one who did this. Heron of Alexandria. In the news much today, Alexandria was the capital of ancient Egypt.

He was the first one to write this method down a long time ago. Though it's believed that even before Heron, the Babylonians know how to do it. So you start with a guess, g. Any old guess will do. Then you say, is g times g close enough to x? If so, you stop. Say OK, I've got a good enough approximation of the answer.

If it's not, you create a new guess by averaging g and x divided by g. So g new is going to be g old plus x divided by g old, over 2. And then using this new guess, you go back to step 2.

So let's quickly run through an example of this. We can start with this pretty easily. We'll take a guess. Let's say g equals 3. So we look at three times 3. 9. And we say, is that good enough?

Well, let's say we're looking for the root of 25. I guess I should have started with the problem statement. Sorry about that. Well, 9 is probably not close enough to 25 that we're happy. May be good enough for government work, but not for most other

purposes.

So we'll reset g, and we'll set g to 3 plus 25 over 3. All of that over 2. Which equals 5.6666 et cetera. All right.

So now we'll multiply that by itself. And that gets to be about 32.04. Close enough to 25? Probably not.

So we'll take another step. And we'll set g equal to-- well, when we're done with it all, I'm not going to bore you with writing the formula again. It'll be 5.04. If we square that, it's 25.4. We decide that's close enough to 25, and we're done.

What we say at this point is that the algorithm-- and that's an important word. An algorithm is a description of how to perform a computation. We say that the algorithm has converged. Which is a fancy way to say it's halted.

What we've got here, if you think about it, is a set of instructions. Steps that can be executed and a flow of control. The order in which we execute them.

So if we look at this, there's a default order of execution-- 1, 2, 3, 4. But then there's the go back to step 2 and start over. And there's a termination condition. It tells us when to stop. And of course, that's important.

I've always been amused, if you look at a shampoo bottle, you'll see an algorithm that says something like lather, rinse, repeat. And if you follow it literally, you never get to stop. Which I suppose make sense if you're selling shampoo, because people use a lot of it. But really, there ought to be some termination condition there.

OK. So now, how do we capture this idea of a recipe in a mechanical process? One way would be to design a machine specifically to do square roots. So if I knew how to design circuits, which I don't, I could sit down-- probably many of you could sit down-- and design a circuit that would implement this algorithm.

And in fact, that's more or less what you'll find in a cheap four function calculator that does square roots. Not quite this algorithm, but a similar algorithm is just part of

the circuitry to go compute that.

And in fact, this used to be the way that all computers worked. So the initial computers were what are called fixed program computers. They were designed to do very specific things, and that's what they did.

So for example, one of the very first computers, designed in 1941, by Atanasoff and Berry, solved systems of linear equations for the purpose of plotting artillery trajectories. And that's all it did. If you wanted to balance your bank account with this computer, you couldn't do it. But you could figure out how to drop an artillery shell somewhere.

Also during World War II, Alan Turing built a machine specifically designed for breaking the German enigma code. Actually a fascinating story of science how that was built. But again, that was all it could do.

These computers were useful, but only in a very limited way. The big breakthrough, the thing it made computation really important to society, was the invention of the stored program computer. It took people quite a while to figure this out. But once they did, it seems obvious.

The basic notion of a stored program computer is that the instructions are the same as data. So now, there is no distinction between the program that implements the algorithm and the data on which that program operates. So there's no difference between the input of 25, part of the data, and the steps of the algorithm used to do that.

Once that was possible, the machines became infinitely flexible. You could change the program anytime you wanted. And furthermore, programs could produce programs. Because programs can produce data.

And if program and data are the same thing, that means programs can produce programs. And we were off and running. And that's really what made computers what they are today.

Once this became clear as the paradigm for computers, people began to think of the computer itself as a program. And in particular, as a kind of program called an interpreter. And we'll get to more on this later today.

An interpreter is a program that can execute any legal set of instructions. And consequently, can be used to describe and accomplish anything you can do with a computer.

So roughly speaking, this is what a stored program computer looks like. This is 6004 in 40 seconds. It's got memory. Lots of it today. A control unit that basically tells it what to do. For example, fetch some data from memory, put some data into memory, send some output to a screen, all of those kinds of things.

What for historical reasons we call the arithmetic logic unit, this is, in some sense, the brains of the computer. The thing that actually does computations. An accumulator, which is part of the ALU that stores results. And a bunch of input and output devices. The things that we actually see when we use a computer. And that's it.

And again, the key thing to notice is, there's only one kind of memory. There's not a memory for program and a memory for data. There's just the memory.

The nice thing to think about here is, given a small set of instructions, you can then build any kind of program you want. So typically, the computers have a very small number of built-in instructions. Order of dozens, and that's it.

And by combining those instructions in very clever ways, you can do arbitrarily complex things. In much the same way a good chef can take a very small number of ingredients, and from those, produce a variety of interesting edibles.

Alan Turing, in the 1930s-- very famous British mathematician of whom you will hear more-- showed that, in fact, there were six primitive instructions. Each of which operated on one bit of information.

And with those six primitive instructions, you could do anything that could be done

with a computer. Kind of amazing. It was six instructions. There were things like read, write, plus-- I don't know, maybe minus. I forget what they were. And that was it. That's all you needed.

We will not make you write programs using only six instructions. We will give you a much larger set. But still, it's really quite remarkable. It's what makes programming such an amazing endeavor.

OK. So what instructions will you be using? Well, that's what a programming language does. So a programming language provides a set of primitive instructions. A set of primitive control structures. So instructions and mechanisms for controlling the order in which they get executed. And that's all. And then you can do whatever you want with them.

And what distinguishes one programming language from another is what these things are. What are your instructions? What of your flow of control? And how do you combine them? What are the combining mechanisms? And in fact, it's the combining mechanisms more than anything else that separate one language from another.

The most amazing thing about programming-- and this has its good side and its bad side, and it's something you need to remember as you do the problem sets-- is that the computer will always do exactly what you tell what to do. It's remarkable.

You don't have any friends who will do whatever you tell them to do. I can tell you my children certainly don't do whatever I tell them to do. And my wife doesn't either. Sometimes she probably thinks I do whatever she tells me to do.

But a computer will do what you tell it to do. So that's very empowering. It's also very annoying. Because it means if your program doesn't work, it's your own darn fault. You got nobody else to blame but yourself. Because it's not the computer's fault. You may want to curse the computer, but you shouldn't. It's just doing what you told it to. So be careful what you wish for.

All right. The programming language we're going to use in 600 is Python. It's a

relatively recent addition to the universe of languages. I want to emphasize that this course is not about learning Python. I will spend relatively little time in the lectures telling you about Python.

It's about computational methods, is what this course is really about. And Python is merely a teaching tool. Once you learn to program in Python, it's easy to learn to program in another language. It's a very easily transferable skill.

If we think about what defines any programming language, it's got a syntax, a static semantics, and a semantics. Are any of you here linguistics majors? Not a one. All right. Then I can make up whatever I want about these terms and maybe you'll believe me.

All right. So the syntax tells us which sequences of characters and symbols constitute a well-formed string. So it would tell us, maybe, that we could write something like x equals 3 plus 4. And that's syntactically correct. It's well-formed. It might also tell us that x equals 3 blank 4 is not syntactically correct. It's not a legal string.

So by analogy with English, the syntax describes which strings of words constitute well-formed sentences. Well-formed. Not necessarily meaningful. So it would tell you that some sentence like Susan is building is syntactically well-formed. It may not be very sensible.

The static semantics tells us which well-formed strings have a meaning. That are which strings are meaningful. So you can think about that as also making sense. So in Python, it might tell us that some strings which are syntactically fine don't mean anything.

So for example, it might tell us that the string 3 divided by the character string abc is syntactically well-formed because it's value operator value. Sort of like noun verb noun is syntactically well-formed in English. But it would tell us that there's no real meaning to this. Dividing a number by a string doesn't mean anything.

And so you would get an error message saying the syntax is OK, but the static

semantics is broken. So for example, in English , the sentence I are big is somehow syntactically well-formed-- noun verb noun-- but we might say it fails the static semantic test. We don't want to assign a meaning to it.

The semantics of the language looks only at the strings that are both syntactically correct and static semantically correct, and assigns a real meaning to them. In natural language, sentences can be ambiguous.

So one of my favorites, when I have to write a recommendation letter for a student that maybe I don't think is so good, I might say something like I cannot praise this student too highly. Well, you can interpret that however you want. It keeps me from getting sued, but I can also claim, well, I don't like the student at all.

And English is full of those things. Programming languages, in contrast, are designed so that every well-formed program has exactly one meaning. There's no ambiguity. So you can't typically talk of a program as having a semantic error. If it is well-formed, it means something, and that's what it means.

On the other hand, it's easy to talk about a program meaning something other than you wanted it to mean. And you will discover in the problem sets, most of the time the programs don't mean what you want them to mean. That is to say, when you run them, they don't give you the correct answer. And then you will go through this process of debugging them and learning how to do it.

So what might happen when we write a program that doesn't do what we want it to do? It might crash. By that, we mean stop running and produce some palpable indication that it has done so.

So you've all used programs that have crashed, right? You sat there using your email program or Word, or PowerPoint, or something. And suddenly, it just goes away. And you get a message on your screen and an invitation to send Apple or Microsoft a file explaining what went wrong so they can fix it.

In a properly designed computing system, when one program crashes, it does not

damage the overall system. So you'd like it to just be local.

What else might it do? It might never stop. Now, if you have no idea how long a program is supposed to run, this can be hard to diagnose. But again, I'm sure you've all run into this. I've certainly run into it.

Every once in a while I'll say, try and write a PowerPoint file. And it'll just sit there. Or I'll try to read a file and it'll just sit there and never finish the job. Or I don't have enough patience. But probably it would never have finished it.

Again, you will all write programs that do this. It's a good idea to know how long you expect your programs to run, so that you can recognize this. Typically, we say that these programs have in them an infinite loop. And we'll talk about that when we get to flow of control.

Finally, a program might run to completion and produce the wrong answer. These problems are kind of in ascending order of badness. If it crashes, at least you know that something has gone wrong.

An infinite loop can be very annoying, because you just wait for a long time. But the worst thing that happens is when you think everything is good and it's not. There have been lots of examples of this. This is the sort of thing that costs lives.

There was a radiation therapy machine that produced the wrong dosage of radiation and actually killed quite a few people, because they put in the correct input, and it would dose the patient with radiation. And a fatal dose of radiation. That's a really bad mistake.

There are buildings that collapse because people run programs that do the structural engineering, and the programs give the wrong answer. Lots of bad things can happen. So one of the things we're going to spend time on this term is, what you can do to avoid writing programs that have this rather unpleasant property.

How do you test them? How do you write them in such a way that this is the least likely event? That's not what you want to happen.

OK. Some programming languages give you a lot of help in avoiding these things. Python is kind of mediocre in that respect. It's not the best. It's not the worst. It's somewhere in the middle.

Because what you'd like is a program with very rigorous static semantics, such that if you pass those tests, it has a high probability of behaving as expected. So for example, it's a good thing that Python doesn't allow you to do this. Because who knows what that's going to do? Something weird.

You'd rather be told no, you can't write that. And then you have to write something that's more obviously meaningful. Rather than it just making up an interpretation.

As we will see going forward, Python is not, for example, as good as Java is at weeding out meaningless things. Or things that have surprising meanings. On the other hand, it's better than C. So kind of in the middle as these programming languages go.

Why do we use Python in this course if it's not the best in that respect? It's got several good features. One of them is, it's easy to learn. It's much less complicated than, say, Java. So the learning curve is much steeper. That's a good thing. You get up to speed faster.

It's very widely used today in a lot of areas of science, particularly the life sciences. It has probably become the most popular language in biology and the other life sciences. And therefore, for those of you who have careers in that area, it's the most useful language to know. It's also widely used in other areas as well.

It is easier to debug than most languages. And the reason it's easier to debug than most languages, or than many, is it's an interpreted language. So you'll remember, I talked about a computer as an interpreter. Something that you feed in a bunch of instructions, called the source code.

You do some checking. And then it executes the instructions, including the flow of control instructions. Produces some output. The nice thing that goes on there is if

something untoward happens, the interpreter can describe in the language of the source code what went wrong. The source code is the code that you wrote.

On the other hand, the way a compiler works is, you take the source code, you check it, but then you translate it into another language called the object code. This is a language closer to the language that the computer, the hardware, knows how to interpret.

Then the hardware interpreter interprets the compiled code, the object code, and produces output. And the problem here is if something goes wrong, it wants to give you an error message in terms of the object code, which you've never seen in your life. And that can make it very obscure.

So the advantage? Why do we have compilers? Typically, compiled languages are more efficient. Because they go through this extra step, they take less time to run those programs. You can compile Python as well, if you want to get an efficient version. But it's not designed under that assumption. And so, it works well when it's interpreted, which is why we use it.