PROFESSOR: Good morning. Try it again. Good morning.

STUDENTS: Good morning.

PROFESSOR: Thank you. This is 6.00, also known as Introduction to Computer Science and Programming. My name is Eric Grimson, I have together Professor John Guttag over here, we're going to be lecturing the course this term. I want to give you a heads up; you're getting some serious firepower this term. John was department head for ten years, felt like a century, and in course six, I'm the current department head in course six. John's been lecturing for thirty years, roughly. All right, I'm the young guy, I've only been lecturing for twenty-five years. You can tell, I have less grey hair than he does. What I'm trying to say to you is, we take this course really seriously. We hope you do as well. But we think it's really important for the department to help everybody learn about computation, and that's what this course is about.

What I want to do today is three things: I'm going to start-- actually, I shouldn't say start, I'm going to do a little bit of administrivia, the kinds of things you need to know about how we're going to run the course. I want to talk about the goal of the course, what it is you'll be able to do at the end of this course when you get through it, and then I want to begin talking about the concepts and tools of computational thinking, which is what we're primarily going to focus on here. We're going to try and help you learn how to think like a computer scientist, and we're going to begin talking about that towards the end of this lecture and of course throughout the rest of the lectures that carry on.

Right, let's start with the goals. I'm going to give you goals in two levels. The strategic goals are the following: we want to help prepare freshmen and sophomores who are interested in majoring in course six to get an easy entry into the department, especially for those students who don't have a lot of prior programming experience. If you're in that category, don't panic, you're going to get it. We're going to help you ramp in and you'll certainly be able to start the course six curriculum and do just fine and still finish on target. We don't expect everybody to be a course six major, contrary to popular opinion, so for those are you not in that category, the second thing we want to do is we want to help students who don't plan to major in course six to feel justifiably confident in their ability to write and read small pieces of code.

For all students, what we want to do is we want to give you an understanding of the role computation can and

cannot play in tackling technical problems. So that you will come away with a sense of what you can do, what you can't do, and what kinds of things you should use to tackle complex problems.

And finally, we want to position all students so that you can easily, if you like, compete for things like your office and summer jobs. Because you'll have an appropriate level of confidence and competence in your ability to do computational problem solving. Those are the strategic goals.

Now, this course is primarily aimed at students who have little or no prior programming experience. As a consequence, we believe that no student here is under-qualified for this course: you're all MIT students, you're all qualified to be here. But we also hope that there aren't any students here who are over-qualified for this course. And what do I mean by that? If you've done a lot prior programming, this is probably not the best course for you, and if you're in that category, I would please encourage you to talk to John or I after class about what your goals are, what kind of experience you have, and how we might find you a course that better meets your goals.

Second reason we don't want over-qualified students in the class, it sounds a little nasty, but the second reason is, an over-qualified student, somebody who's, I don't know, programmed for Google for the last five years, is going to have an easy time in this course, but we don't want such a student accidentally intimidating the rest of you. We don't want you to feel inadequate when you're simply inexperienced. And so, it really is a course aimed at students with little or no prior programming experience. And again, if you're not in that category, talk to John or I after class, and we'll help you figure out where you might want to go.

OK. Those are the top-level goals of the course. Let's talk sort of at a more tactical level, about what do we want you to know in this course. What we want you to be able to do by the time you leave this course? So here are the skills that we would like you to acquire. Right, the first skill we want you to acquire, is we want you to be able to use the basic tools of computational thinking to write small scale programs. I'm going to keep coming back to that idea, but I'm going to call it computational thinking. And that's so you can write small pieces of code. And small is not derogatory here, by the way, it just says the size of things you're going to be able to do.

Second skill we want you to have at the end of this course is the ability to use a vocabulary of computational tools in order to be able to understand programs written by others. So you're going to be able to write, you're going to be able to read.

This latter skill, by the way, is incredibly valuable. Because you won't want to do everything from scratch yourself, you want to be able to look at what is being created by somebody else and understand what is inside of there, whether it works correctly and how you can build on it. This is one of the few places where plagiarism is an OK thing. It's not bad to, if you like, learn from the skills of others in order to create something you want to write. Although we'll come back to plagiarism as a bad thing later on.

Third thing we want you to do, is to understand the fundamental both capabilities and limitations of computations, and the costs associated with them. And that latter statement sounds funny, you don't think of computations having limits, but they do. There're some things that cannot be computed. We want you to understand where those limits are. So you're going to be able to understand abilities and limits.

And then, finally, the last tactical skill that you're going to get out of this course is you're going to have the ability to map scientific problems into a computational frame. So you're going to be able to take a description of a problem and map it into something computational.

Now if you think about it, boy, it sounds like grammar school. We're going to teach you to read, we're going to teach you to write, we're going to teach you to understand what you can and cannot do, and most importantly, we're going to try and give you the start of an ability to take a description of a problem from some other domain, and figure out how to map it into that domain of computation so you can do the reading and writing that you want to do.

OK, in a few minutes we're going to start talking then about what is computation, how are we going to start building those tools, but that's what you should take away, that's what you're going to gain out of this course by the time you're done.

Now, let me take a sidebar for about five minutes to talk about course administration, the administrivia, things that we're going to do in the course, just so you know what the rules are. Right, so, class is two hours of lecture a week. You obviously know where and you know when, because you're here. Tuesdays and Thursdays at 11:00. One hour of recitation a week, on Fridays, and we'll come back in a second to how you're going to get set up for that. And nine hours a week of outside-the-class work. Those nine hours are going to be primarily working on problem sets, and all the problems sets are going to involve programming in Python, which is the language we're going to be using this term.

Now, one of the things you're going to see is the first problem sets are pretty easy. Actually, that's probably wrong, John, right? They're very easy. And we're going to ramp up. By the time you get to the end of the term, you're going to be dealing with some fairly complex things, so one of the things you're going to see is, we're going to make heavy use of libraries, or code written by others. It'll allow you to tackle interesting problems I'll have you to write from scratch, but it does mean that this skill here is going to be really valuable. You need to be able to read that code and understand it, as well as write your own.

OK. Two quizzes. During the term, the dates have already been scheduled. John, I forgot to look them up, I think it's October 2nd and November 4th, it'll be on the course website. My point is, go check the course website, which

by the way is right there. If you have, if you know you have a conflict with one of those quiz dates now, please see John or I right away. We'll arrange something ahead of time. But if you-- The reason I'm saying that is, you know, you know that you're getting married that day for example, we will excuse you from the quiz to get married. We'll expect you come right back to do the quiz by the way, but the-- Boy, tough crowd. All right. If you have a conflict, please let us know.

Second thing is, if you have an MIT documented special need for taking quizzes, please see John or I well in advance. At least two weeks before the quiz. Again, we'll arrange for this, but you need to give us enough warning so that we can deal with that.

OK, the quizzes are open book. This course is not about memory. It's not how well you can memorize facts: in fact, I think both John and I are a little sensitive to memory tests, given our age, right John? This is not about how you memorize things, it's about how you think. So they're open note, open book. It's really going to test your ability to think.

The grades for the course will be assigned roughly, and I use the word roughly because we reserve the right to move these numbers around a little bit, but basically in the following percentages: 55% of your grade comes from the problem sets, the other 45% come from the quizzes. And I should've said there's two quizzes and a final exam. I forgot, that final exam during final period. So the quiz percentages are 10%, 15%, and 20%. Which makes up the other 45%.

OK. Other administrivia. Let me just look through my list here. First problem set, problem set zero, has already been posted. This is a really easy one. We intend it to be a really easy problem set. It's basically to get you to load up Python on your machine and make sure you understand how to interact with it.

The first problem set will be posted shortly, it's also pretty boring-- somewhat like my lectures but not John's-- and that means, you know, we want you just to get going on things. Don't worry, we're going to make them more interesting as you go along.

Nonetheless, I want to stress that none of these problems sets are intended to be lethal. We're not using them to weed you out, we're using them to help you learn. So if you run into a problem set that just, you don't get, all right? Seek help. Could be psychiatric help, could be a TA. I recommend the TA. My point being, please come and talk to somebody. The problems are set up so that, if you start down the right path, it should be pretty straight-forward to work it through. If you start down a plausible but incorrect path, you can sometimes find yourself stuck in the weeds somewhere, and we want to bring you back in. So part of the goal here is, this should not be a grueling, exhausting kind of task, it's really something that should be helping you learn the material. If you need help, ask John, myself, or the TAs. That's what we're here for.

OK. We're going to run primarily a paperless subject, that's why the website is there. Please check it, that's where everything's going to be posted in terms of things you need to know. In particular, please go to it today, you will find a form there that you need to fill out to register for, or sign up for rather, a recitation.

Recitations are on Friday. Right now, we have them scheduled at 9:00, 10:00, 11:00, 12:00, 1:00, and 2:00. We may drop one of the recitations, just depending on course size, all right? So we reserve the right, unfortunately, to have to move you around. My guess is that 9:00 is not going to be a tremendously popular time, but maybe you'll surprise me. Nonetheless, please go in and sign up. We will let you sign up for whichever recitation makes sense for you. Again, we reserve the right to move people around if we have to, just to balance load, but we want you to find something that fits your schedule rather than ours.

OK. Other things. There is no required text. If you feel exposed without a text book, you really have to have a textbook, you'll find one recommended-- actually I'm going to reuse that word, John, at least suggest it, on the course website. I don't think either of us are thrilled with the text, it's the best we've probably found for Python, it's OK. If you need it, it's there. But we're going to basically not rely on any specific text.

Right. Related to that: attendance here is obviously not mandatory. You ain't in high school anymore. I think both of us would love to see your smiling faces, or at least your faces, even if you're not smiling at us every day. Point I want to make about this, though, is that we are going to cover a lot of material that is not in the assigned readings, and we do have assigned readings associated with each one of these lectures. If you choose not to show up today-- or sorry, you did choose to show up today, if you choose not to show up in future days-- we'll understand, but please also understand that the TAs won't have a lot of patience with you if you're asking a question about something that was either covered in the readings, or covered in the lecture and is pretty straight forward. All right? We expect you to behave responsibly and we will as well. All right.

I think the last thing I want to say is, we will not be handing out class notes. Now this sounds like a draconian measure; let me tell you why. Every study I know of, and I suspect every one John knows, about learning, stresses that students learn best when they take notes. Ironically, even if they never look at them. OK. The process of writing is exercising both halves of your brain, and it's actually helping you learn, and so taking notes is really valuable thing. Therefore we're not going to distribute notes. What we will distribute for most lectures is a handout that's mostly code examples that we're going to do. I don't happen to have one today because we're not going to do a lot of code. We will in future. Those notes are going to make no sense, I'm guessing, outside of the lecture, all right? So it's not just, you can swing by 11:04 and grab a copy and go off and catch some more sleep. What we recommend is you use those notes to take your own annotations to help you understand what's going on, but we're not going to provide class notes. We want you to take your own notes to help you, if you like, spur

your own learning process.

All right. And then finally, I want to stress that John, myself, all of the staff, our job is to help you learn. That's what we're here for. It's what we get excited about. If you're stuck, if you're struggling, if you're not certain about something, please ask. We're not mind readers, we can't tell when you're struggling, other than sort of seeing the expression on your face, we need your help in identifying that. But all of the TAs, many of whom are sitting down in the front row over here, are here to help, so come and ask. At the same time, remember that they're students too. And if you come and ask a question that you could have easily answered by doing the reading, coming to lecture, or using Google, they're going to have less patience. But helping you understand things that really are a conceptual difficulty is what they're here for and what we're here for, so please come and talk to us.

OK. That takes care of the administrivia preamble. John, things we add?

PROFESSOR GUTTAG: Two more quick things. This semester, your class is being videotaped for OpenCourseware. If any of you don't want your image recorded and posted on the web, you're supposed to sit in the back three rows.

PROFESSOR GRIMSON: Ah, thank you. I forgot.

PROFESSOR GUTTAG: --Because the camera may pan. I think you're all very good-looking and give MIT a good image, so please, feel free to be filmed.

PROFESSOR GRIMSON: I'll turn around, so if you want to, you know, move to the back, I won't see who moves. Right. Great. Thank you, John.

PROFESSOR GUTTAG: So that, the other thing I want to mention is, recitations are also very important. We will be covering material in recitations that're not in the lectures, not in the reading, and we do expect you to attend recitations.

PROFESSOR GRIMSON: Great. Thanks, John. Any questions about the administrivia? I know it's boring, but we need to do it so you know what the ground rules are.

Good. OK. Let's talk about computation. As I said, our strategic goal, our tactical goals, are to help you think like a computer scientist. Another way of saying it is, we want to give you the skill so that you can make the computer do what you want it to do. And we hope that at the end of the class, every time you're confronted with some technical problem, one of your first instincts is going to be, "How do I write the piece of code that's going to help me solve that?"

So we want to help you think like a computer scientist. All right. And that, is an interesting statement. What does it mean, to think like a computer scientist? Well, let's see. The primary knowledge you're going to take away from this course is this notion of computational problem solving, this ability to think in computational modes of thought. And unlike in a lot of introductory courses, as a consequence, having the ability to memorize is not going to help you. It's really learning those notions of the tools that you want to use. What in the world does it mean to say computational mode of thought? It sounds like a hifalutin phrase you use when you're trying to persuade a VC to fund you. Right. So to answer this, we really have to ask a different question, a related question; so, what's computation?

It's like a strange statement, right? What is computation? And part of the reason for putting it up is that I want to, as much as possible, answer that question by separating out the mechanism, which is the computer, from computational thinking. Right. The artifact should not be what's driving this. It should be the notion of, "What does it mean to do computation?"

Now, to answer that, I'm going to back up one more level. And I'm going to pose what sounds like a philosophy question, which is, "What is knowledge?" And you'll see in about two minutes why I'm going to do this. But I'm going to suggest that I can divide knowledge into at least two categories. OK, and what is knowledge? And the two categories I'm going to divide them into are declarative and imperative knowledge.

Right. What in the world is declarative knowledge? Think of it as statements of fact. It's assertions of truth. Boy, in this political season, that's a really dangerous phrase to use, right? But it's a statement of fact. I'll stay away from the political comments. Let me give you an example of this. Right. Here's a declarative statement. The square root of x is that y such that y squared equals x, y's positive. You all know that. But what I want you to see here, is that's a statement of fact. It's a definition. It's an axiom. It doesn't help you find square roots. If I say x is 2, I want to know, what's the square root of 2, well if you're enough of a geek, you'll say 1.41529 or whatever the heck it is, but in general, this doesn't help you find the square root. The closest it does is it would let you test. You know, if you're wandering through Harvard Square and you see an out-of-work Harvard grad, they're handing out examples of square roots, they'll give you an example and you can test it to see, is the square root of 2, 1.41529 or whatever. I don't even get laughs at Harvard jokes, John, I'm going to stop in a second here, all right? All right, so what am I trying to say here? It doesn't -- yeah, exactly. We're staying away from that, really quickly, especially with the cameras rolling.

All right. What am I trying to say? It tells you how you might test something but it doesn't tell you how to.

And that's what imperative knowledge is. Imperative knowledge is a description of how to deduce something. So let me give you an example of a piece of imperative knowledge. All right, this is actually a very old piece of

imperative knowledge for computing square roots, it's attributed to Heron of Alexandria, although I believe that the Babylonians are suspected of knowing it beforehand. But here is a piece of imperative knowledge. All right? I'm going to start with a guess, I'm going to call it g. And then I'm going to say, if g squared is close to x, stop. And return g. It's a good enough answer. Otherwise, I'm going to get a new guess by taking g, x over g, adding them, and dividing by two. Then you take the average of g and x over g. Don't worry about how came about, Heron found this out. But that gives me a new guess, and I'm going to repeat.

That's a recipe. That's a description of a set of steps. Notice what it has, it has a bunch of nice things that we want to use, right? It's a sequence of specific instructions that I do in order. Along the way I have some tests, and depending on the value of that test, I may change where I am in that sequence of instructions. And it has an end test, something that tells me when I'm done and what the answer is. This tells you how to find square roots. it's how-to knowledge. It's imperative knowledge.

All right. That's what computation basically is about. We want to have ways of capturing this process. OK, and that leads now to an interesting question, which would be, "How do I build a mechanical process to capture that set of computations?" So I'm going to suggest that there's an easy way to do it-- I realized I did the boards in the wrong order here-- one of the ways I could do it is, you could imagine building a little circuit to do this. If I had a couple of elements of stored values in it, I had some wires to move things around, I had a little thing to do addition, little thing to do division, and a something to do the testing, I could build a little circuit that would actually do this computation.

OK. That, strange as it sounds, is actually an example of the earliest computers, because the earliest computers were what we call fixed-program computers, meaning that they had a piece of circuitry designed to do a specific computation. And that's what they would do: they would do that specific computation. You've seen these a lot, right? A good example of this: calculator. It's basically an example of a fixed-program computer. It does arithmetic. If you want play video games on it, good luck. If you want to do word processing on it, good luck. It's designed to do a specific thing. It's a fixed-program computer.

In fact, a lot of the other really interesting early ones similarly have this flavor, to give an example: I never know how to pronounce this, Atanasoff, 1941. One of the earliest computational things was a thing designed by a guy named Atanasoff, and it basically solved linear equations. Handy thing to do if you're doing 1801, all right, or 1806, or whatever you want to do those things in. All it could do, though, was solve those equations.

One of my favorite examples of an early computer was done by Alan Turing, one of the great computer scientists of all time, called the bombe, which was designed to break codes. It was actually used during WWII to break German Enigma codes. And what it was designed to do, was to solve that specific problem.

The point I'm trying to make is, fixed-program computers is where we started, but it doesn't really get us to where we'd like to be. We want to capture this idea of problem solving. So let's see how we'd get there. So even within this framework of, given a description of a computation as a set of steps, in the idea that I could build a circuit to do it, let me suggest for you what would be a wonderful circuit to build.

Suppose you could build a circuit with the following property: the input to this circuit would be any other circuit diagram. Give it a circuit diagram for some computation, you give it to the circuit, and that circuit would wonderfully reconfigure itself to act like the circuits diagram. Which would mean, it could act like a calculator. Or, it could act like Turing's bombe. Or, it could act like a square root machine.

So what would that circuit look like? You can imagine these tiny little robots wandering around, right? Pulling wires and pulling out components and stacking them together. How would you build a circuit that could take a circuit diagram in and make a machine act like that circuit? Sounds like a neat challenge.

Let me change the game slightly. Suppose instead, I want a machine that can take a recipe, the description of a sequence of steps, take that as its input, and then that machine will now act like what is described in that recipe. Reconfigure itself, emulate it, however you want to use the words, it's going to change how it does the computation.

That would be cool. And that exists. It's called an interpreter. It is the basic heart of every computer. What it is doing, is saying, change the game. This is now an example of a stored-program computer. What that means, in a stored-program computer, is that I can provide to the computer a sequence of instructions describing the process I want it to execute. And inside of the machine, and things we'll talk about, there is a process that will allow that sequence to be executed as described in that recipe, so it can behave like any thing that I can describe in one of those recipes.

All right. That actually seems like a really nice thing to have, and so let me show you what that would basically look like. Inside of a stored-program computer, we would have the following: we have a memory, it's connected to two things; control unit, in what's called an ALU, an arithmetic logic unit, and this can take in input, and spit out output, and inside this stored-program computer, excuse me, you have the following: you have a sequence of instructions. And these all get stored in there. Notice the difference. The recipe, the sequence of instructions, is actually getting read in, and it's treated just like data. It's inside the memory of the machine, which means we have access to it, we can change it, we can use it to build new pieces of code, as well as we can interpret it. One other piece that goes into this computer-- I never remember where to put the PC, John, control? ALU? Separate? I'll put it separate-- you have a thing called a program counter.

And here's the basis of the computation. That program counter points to some location in memory, typically to the

first instruction in the sequence. And those instructions, by the way, are very simple: they're things like, take the value out of two places in memory, and run them through the multiplier in here, a little piece of circuitry, and stick them back into someplace in memory. Or take this value out of memory, run it through some other simple operation, stick it back in memory. Having executed this instruction, that counter goes up by one and we move to the next one. We execute that instruction, we move to the next one. Oh yeah, it looks a whole lot like that.

Some of those instructions will involve tests: they'll say, is something true? And if the test is true, it will change the value of this program counter to point to some other place in the memory, some other point in that sequence of instructions, and you'll keep processing. Eventually you'll hopefully stop, and a value gets spit out, and you're done.

That's the heart of a computer. Now that's a slight misstatement. The process to control it is intriguing and interesting, but the heart of the computer is simply this notion that we build our descriptions, our recipes, on a sequence of primitive instructions. And then we have a flow of control. And that flow of control is what I just described. It's moving through a sequence of instructions, occasionally changing where we are as we move around.

OK. The thing I want you to take away from this, then, is to think of this as, this is, if you like, a recipe. And that's really what a program is. It's a sequence of instructions. Now, one of things I left hanging is, I said, OK, you build it out of primitives. So one of the questions is, well, what are the right primitives to use? And one of the things that was useful here is, that we actually know that the set of primitives that you want to use is very straight-forward.

OK, but before I do that, let me drive home this idea of why this is a recipe. Assuming I have a set of primitive instructions that I can describe everything on, I want to know what can I build. Well, I'm going to do the same analogy to a real recipe. So, real recipe. I don't know. Separate six eggs. Do something. Beat until the-- sorry, beat the whites until they're stiff. Do something until an end test is true. Take the yolks and mix them in with the sugar and water-- No. Sugar and flour I guess is probably what I want, sugar and water is not going to do anything interesting for me here-- mix them into something else. Do a sequence of things.

A traditional recipe actually is based on a small set of primitives, and a good chef with, or good cook, I should say, with that set of primitives, can create an unbounded number of great dishes. Same thing holds true in programming. Right. Given a fixed set of primitives, all right, a good programmer can program anything. And by that, I mean anything that can be described in one of these process, you can capture in that set of primitives.

All right, the question is, as I started to say, is, "What are the right primitives?" So there's a little bit of, a little piece of history here, if you like. In 1936, that same guy, Alan Turing, showed that with six simple primitives, anything

that could be described in a mechanical process, it's actually algorithmically, could be programmed just using those six primitives.

Think about that for a second. That's an incredible statement. It says, with six primitives, I can rule the world. With six primitives, I can program anything. A couple of really interesting consequences of that, by the way, one of them is, it says, anything you can do in one programming language, you can do in another programming language. And there is no programming language that is better-- well actually, that's not quite true, there are some better at doing certain kinds of things-- but there's nothing that you can do in C that you can't do in Fortran. It's called Turing compatibility. Anything you can do with one, you can do with another, it's based on that fundamental result.

OK. Now, fortunately we're not going to start with Turing's six primitives, this would be really painful programming, because they're down at the level of, "take this value and write it onto this tape." First of all, we don't have tapes anymore in computers, and even if we did, you don't want to be programming at that level. What we're going to see with programming language is that we're going to use higher-level abstracts. A broader set of primitives, but nonetheless the same fundamental thing holds. With those six primitives, you can do it.

OK. So where are we here? What we're saying is, in order to do computation, we want to describe recipes, we want to describe this sequence of steps built on some primitives, and we want to describe the flow of control that goes through those sequence of steps as we carry on.

So the last thing we need before we can start talking about real programming is, we need to describe those recipes. All right, And to describe the recipes, we're going to want a language. We need to know not only what are the primitives, but how do we make things meaningful in that language. Language. There we go. All right. Now, it turns out there are-- I don't know, John, hundreds? Thousands? Of programming languages? At least hundreds-- of programming languages around.

PROFESSOR JOHN GUTTAG: [UNINTELLIGIBLE]

PROFESSOR ERIC GRIMSON: True. Thank you. You know, they all have, you know, their pluses and minuses. I have to admit, in my career here, I think I've taught in at least three languages, I suspect you've taught more, five or six, John? Both of us have probably programmed in more than those number of languages, at least programmed that many, since we taught in those languages.

One of the things you want to realize is, there is no best language. At least I would argue that, I think John would agree. We might both agree we have our own nominees for worst language, there are some of those. There is no best language. All right? They all are describing different things. Having said that, some of them are better suited

for some things than others.

Anybody here heard of MATLAB Maybe programmed in MATLAB? It's great for doing things with vectors and matrices and things that are easily captured in that framework. But there's some things that are a real pain to do in MATLAB. So MATLAB's great for that kind of thing.

C is a great language for programming things that control data networks, for example.

I happen to be, and John teases me about this regularly, I'm an old-time Lisp programmer, and that's how I was trained. And I happen to like Lisp and Scheme, it's a great language when you're trying to deal with problems where you have arbitrarily structured data sets. It's particularly good at that.

So the point I want to make here is that there's no particularly best language. What we're going to do is simply use a language that helps us understand. So in this course, the language we're going to use is Python. Which is a pretty new language, it's growing in popularity, it has a lot of the elements of some other languages because it's more recent, it inherits things from it's pregenitors, if you like.

But one of the things I want to stress is, this course is not about Python. Strange statement. You do need to know how to use it, but it's not about the details of, where do the semi-colons go in Python. All right? It's about using it to think.

And what you should take away from this course is having learned how to design recipes, how to structure recipes, how to do things in modes in Python. Those same tools easily transfer to any other language. You can pick up another language in a week, couple of weeks at most, once you know how to do Python.

OK. In order to talk about Python and languages, I want to do one last thing to set the stage for what we're going to do here, and that's to talk about the different dimensions of a language. And there're three I want to deal with.

The first one is, whether this is a high-level or low-level language. That basically says, how close are you the guts of the machine? A low-level language, we used to call this assembly programming, you're down at the level of, your primitives are literally moving pieces of data from one location of memory to another, through a very simple operation. A high-level language, the designer has created a much richer set of primitive things. In a high-level language, square root might simply be a primitive that you can use, rather than you having to go over and code it. And there're trade-offs between both.

Second dimension is, whether this is a general versus a targeted language. And by that I mean, do the set of primitives support a broad range of applications, or is it really aimed at a very specific set of applications? I'd argue that MATLAB is basically a targeted language, it's targeted at matrices and vectors and things like that.

And the third one I want to point out is, whether this is an interpreted versus a compiled language. What that basically says is the following: in an interpreted language, you take what's called the source code, the thing you write, it may go through a simple checker but it basically goes to the interpreter, that thing inside the machine that's going to control the flow of going through each one of the instructions, and give you an output. So the interpreter is simply operating directly on your code at run time.

In a compiled language, you have an intermediate step, in which you take the source code, it runs through what's called a checker or a compiler or both, and it creates what's called object code. And that does two things: one, it helps catch bugs in your code, and secondly it often converts it into a more efficient sequence of instructions before you actually go off and run it. All right?

And there's trade-offs between both. I mean, an interpreted language is often easier to debug, because you can still see your raw code there, but it's not always as fast. A compiled language is usually much faster in terms of its execution. And it's one of the things you may want to trade off.

Right. In the case of Python, it's a high-level language. I would argue, I think John would agree with me, it's basically a general-purpose language. It happens to be better suited for manipulating strings than numbers, for example, but it's really a general-purpose language. And it's primarily-- I shouldn't say primarily, it is an interpreted language. OK?

As a consequence, it's not as good as helping debug, but it does let you-- sorry, that's the wrong way of saying-- it's not as good at catching some things before you run them, it is easier at some times in debugging as you go along on the fly.

OK. So what does Python look like? In order to talk about Python-- actually, I'm going to do it this way-- we need to talk about how to write things in Python. Again, you have to let me back up slightly and set the stage.

Our goal is to build recipes. You're all going to be great chefs by the time you're done here. All right? Our goal is to take problems and break them down into these computational steps, these sequence of instructions that'll allow us to capture that process.

To do that, we need to describe: not only, what are the primitives, but how do we capture things legally in that language, and interact with the computer? And so for that, we need a language. We're about to start talking about the elements of the language, but to do that, we also need to separate out one last piece of distinction. Just like with a natural language, we're going to separate out syntax versus semantics.

So what's syntax? Syntax basically says, what are the legal expressions in this language? Boy, my handwriting is

atrocious, isn't it? There's a English sequence of words. It's not since syntactically correct, right? It's not a sentence. There's no verb in there anywhere, it's just a sequence of nouns. Same thing in our languages. We have to describe how do you put together legally formed expressions. OK? And as we add constructs to the language, we're going to talk about.

Second thing we want to talk about very briefly as we go along is the semantics of the language. And here we're going to break out two pieces; static semantics and full semantics. Static semantics basically says which programs are meaningful. Which expressions make sense. Here's an English sentence. It's syntactically correct. Right? Noun phrase, verb, noun phrase. I'm not certain it's meaningful, unless you are in the habit of giving your furniture personal names.

What's the point? Again, you can have things that are syntactically legal but not semantically meaningful, and static semantics is going to be a way of helping us decide what expressions, what pieces of code, actually have real meaning to it. All right?

The last piece of it is, in addition to having static semantics, we have sort of full semantics. Which is, what does the program mean? Or, said a different way, what's going to happen when I run it? That's the meaning of the expression. That's what you want. All right? You want to know, what's the meaning of this piece of code? When I run it, what's going to happen? That's what I want to build. The reason for pulling this out is, what you're going to see is, that in most languages, and certainly in Python-- we got lots of help here-- all right, Python comes built-in with something that will check your static, sorry, your syntax for you. And in fact, as a sidebar, if you turn in a problem set that is not syntactically correct, there's a simple button that you push that will check your syntax. If you've turned in a program that's not syntactically correct, the TAs give you a zero. Because it said you didn't even take the time to make sure the syntax is correct. The system will help you find it. In Python, it'll find it, I think one bug at a time, right John? It finds one syntax error at a time, so you have to be a little patient to do it, but you can check that the syntax is right.

You're going to see that we get some help here on the static semantics, and I'm going to do an example in a second, meaning that the system, some languages are better than others on it, but it will try and help you catch some things that are not semantically correct statically. In the case of Python, it does that I think all at run time. I'm looking to you again, John, I think there's no pre-time checks. Its-- sorry?

PROFESSOR JOHN GUTTAG: [UNINTELLIGIBLE]

PROFESSOR ERIC GRIMSON: There is some. OK. Most of them, I think though, are primarily caught at run time, and that's a little bit of a pain because you don't see it until you go and run the code, and there are some, actually we're going to see an example I think in a second where you find it, but you do get some help there.

The problem is, things that you catch here are actually the least worrisome bugs. They're easy to spot, you can't run the program with them there, so you're not going to get weird answers. Not everything is going to get caught in static semantics checking. Some things are going to slide through, and that's actually a bother. It's a problem. Because it says, your program will still give you a value, but it may not be what you intended, and you can't always tell, and that may propagate it's way down through a whole bunch of other computations before it causes some catastrophic failure. So actually, the problem with static semantics is you'd like it to catch everything, you don't always get it. Sadly we don't get much help here. Which is where we'd like it. But that's part of your job.

OK. What happens if you actually have something that's both syntactically correct, and appears to have correct static semantics, and you run it? It could run and give you the right answer, it could crash, it could loop forever, it could run and apparently give you the right answer. And you're not always going to be able to tell. Well, you'll know when it crashes, that doesn't help you very much, but you can't always tell whether something's stuck in an infinite loop or whether it's simply taking a long time to compute. You'd love to have a system that spots that for you, but it's not possible.

And so to deal with this last one, you need to develop style. All right? Meaning, we're going to try to help you with how to develop good programming style, but you need to write in a way in which it is going to be easy for you to spot the places that cause those semantic bugs to occur.

All right. If that sounds like a really long preamble, it is. Let's start with Python. But again, my goal here is to let you see what computation's about, why we need to do it, I'm going to remind you one last time, our goal is to be able to have a set of primitives that we combine into complex expressions, which we can then abstract to treat as primitives, and we want to use that sequence of instructions in this flow of control computing, in order to deduce new information. That imperative knowledge that we talked about right there. So I'm going to start today, we have about five or ten minutes left, I think, in order-- sorry, five minutes left-- in order to do this with some beginnings of Python, and we're going to pick this up obviously, next time, so; simple parts of Python.

In order to create any kinds of expressions, we're going to need values. Primitive data elements. And in Python, we have two to start with; we have numbers, and we have strings. Numbers is what you'd expect. There's a number. There's another number. All right? Strings are captured in Python with an open quote and some sequence of characters followed by a closed quote.

Associated with every data type in Python is a type, which identifies the kind of thing it is. Some of these are obvious. Strings are just a type on their own. But for numbers, for example, we can have a variety of types. So this is something that we would call an integer, or an INT. And this is something we would call a floating point, or a float. Or if you want to think of it as a real number. And there's some others that we can see.

We're going to build up this taxonomy if you like, but the reason it's relevant is, associated with each one of those types is a set of operators that expect certain types of input in order to do their job. And given those types of input, will get back output.

All right. In order to deal with this, let me show you an example, and I hope that comes up, great. What I have here is a Python shell, and I'm going to just show you some simple examples of how we start building expressions. And this'll lead into what you're going to see next time as well as what you're going to do tomorrow. So. Starting with the shell, I can type in expressions.

Actually, let me back up and do this in video. I can type in a number, I get back a number, I can type in a string, I get back the string. Strings, by the way, can have spaces in them, they can have other characters, it's simply a sequence of things, and notice, by the way, that the string five-- sorry, the string's digit five digit two is different than the number 52. The quotes are around them to make that distinction. We're going to see why in a second.

What I'm doing, by the way, here is I'm simply typing in expressions to that interpreter. It's using its set of rules to deduce the value and print them back out. Things I might like to do in here is, I might like to do combinations of things with these. So we have associated with simple things, a set of operations. So for numbers, we have the things you'd expect, the arithmetics. And let me show you some examples of that.

And actually, I'm going to do one other distinction here. What I typed in, things like-- well, let me start this way-- there's an expression. And in Python the expression is, operand, operator, operand, when we're doing simple expressions like this, and if I give it to the interpreter, it gives me back exactly what you'd expect, which is that value. OK?

The distinction I'm going to make is, that's an expression. The interpreter is going to get a value for it. When we start building up code, we're going to use commands. Or statements. Which are actually things that take in a value and ask the computer to do something with it. So I can similarly do this, which is going to look strange because it's going to give me the same value back out, but it actually did a slightly different thing.

And notice, by the way, when I typed it how print showed up in a different color? That's the Python saying, that is a command, that is a specific command to get the value of the expression and print it back out. When we start writing code, you're going to see that difference, but for now, don't worry about it, I just want to plant that idea.

OK. Once we've got that, we can certainly, though, do things like this. Notice the quotes around it. And it treats it as a string, it's simply getting me back the value of that string, 52 times 7, rather than the value of it. Now, once we've got that, we can start doing things. And I'm going to use print here-- if I could type, in order to just to get into

that, I can't type, here we go-- in order to get into the habit. I can print out a string. I can print out-- Ah!-- Here's a first example of something that caught one of my things. This is a static semantic error.

So what went on here? I gave it an expression that had an operand in there. It expected arithmetic types. But I gave two strings. And so it's complaining at me, saying, you can't do this. I don't know how to take two strings and multiply them together. Unfortunately-- now John you may disagree with me on this one-- unfortunately in Python you can, however, do things like this. What do you figure that's going to do? Look legal? The string three times the number three? Well it happens to give me three threes in a row.

I hate this. I'm sorry, John, I hate this. Because this is overloading that multiplication operator with two different tasks. It's saying, if you give me two numbers, I'll do the right thing. If you give me a number and a string, I'm going to concatenate them together, it's really different operations, but nonetheless, it's what it's going to do.

STUDENT: [UNINTELLIGIBLE]

PROFESSOR ERIC GRIMSON: There you go. You know, there will be a rebuttal phase a little later on, just like with the political debates, and he likes it as a feature, I don't like it, you can tell he's not a Lisp programmer and I am.

All right. I want to do just a couple more quick examples. Here's another one. Ah-ha! Give you an example of a syntax error. Because 52A doesn't make sense. And you might say, wait a minute, isn't that a string, and the answer's no, I didn't say it's a string by putting quotes around it. And notice how the machine responds differently to it. In this case it says, this is a syntax error, and it's actually highlighting where it came from so I can go back and fix it.

All right. Let's do a couple of other simple examples. All right? I can do multiplication. I've already seen that. I can do addition. Three plus five. I can take something to a power, double star, just take three to the fifth power. I can do division, right? Whoa. Right? Three divided by five is zero? Maybe in Bush econom-- no, I'm not going to do any political comments today, I will not say that, all right?

What happened? Well, this is one of the places where you have to be careful. It's doing integer division. So, three divided by five is zero, with a remainder of three. So this is the correct answer. If I wanted to get full, real division, I should make one of them a float. And yes, you can look at that and say, well is that right? Well, up to some level of accuracy, yeah, that's .6 is what I'd like to get out.

All right. I can do other things. In a particular, I have similar operations on strings. OK, I can certainly print out strings, but I can actually add strings together, and just as you saw, I can multiply strings, you can kind of guess what this is going to do. It is going to merge them together into one thing. I want-- I know I'm running you slightly

over, I want to do one last example, it's, I also want to be able to do, have variables to store things. And to do that, in this it says, if I have a value, I want to keep it around, to do that, I can do things like this.

What does that statement do? It says, create a name for a variable-- which I just did there, in fact, let me type it in-- mystring, with an equal sign, which is saying, assign or bind to that name the value of the following expression. As a consequence, I can now refer to that just by its name. If I get the value of mystring, there it is, or if I say, take mystring and add to it the string, mylastname, and print it back out.

So this is the first start of this. What have we done? We've got values, numbers and strings. We have operations to associate with them. I just threw a couple up here. You're going to get a chance to explore them, and you'll see not only are there the standard numerics for strings, there are things like length or plus or other things you can do with them. And once I have values, I want to get a hold of them so I can give them names. And that's what I just did when I bound that. I said, use the name mystring to be bound to or have the value of Eric, so I can refer to it anywhere else that I want to use it.

And I apologize for taking you over, we'll come back to this next time, please go to the website to sign up for recitation for tomorrow.