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: Hello, and welcome to 6.01. I'm Denny Freeman. I'm the lecturer.

One thing you should know about today is that there's a single hand-out. You should have picked it up on your way in. It's available at either of the two doors.

What I want to do today in this first lecture is mostly focus on content. But before I do that, since 6.01 is a little bit of an unusual course, I want to give you a little bit of an overview and tell you a little bit about the administration of the course.

6.01 is mostly about modes of reasoning. What we would like you to get out of this course is ways to think about engineering. We want to talk about how do you design, how do you build, how do you construct, how do you debug complicated systems? That's what engineers do, and we're very good at it. And we want to make you very good at it.

We're very good at it. And you know that from your common, everyday experience. Laptops are incredible. As we go through the course, you're going to see that laptops incorporate things from the tiniest, tiniest level, things so small that you can't see them. They're microscopic. The individual transistors are not things that you can see. We develop special tools for you even to be able to visualize them.

And yet, we conglomerate billions of them into a system that works relatively reliably. Now, I realize I'm going out on a limb because when you say things like that, then things always fail. But I'll go out on a limb and say, for the most part, the systems we construct are very reliable.

We'd like you to know how you think about making such a complicated system and making it reliable. We want to tell you about how you would model things. How do you gain insight? How do you get predictability? How do you figure out how something will work before you've built it?

If you're limited to trying out how things work by actually constructing it, you spend a lot of time constructing things that never make it. We want to avoid that by -- where we can -- making a model, analyzing the model, making a prediction from the model, and using that prediction to build a better system on the first try.

We want to tell you about how to augment the physical behavior of a system by putting computation in it. That's a very powerful technique that is increasingly common in anything from a microwave to a refrigerator. We'd like you to know the principles by which to do that.

And we'd like you to be able to build systems that are robust to failure. That's a newer idea. It's something that people are very good at.

If we try to do something, and we make a mistake, we know how to fix it. And often, the fix works. We're less good at doing that in constructing artificial systems, in engineering systems. And we'd like to talk about principles by which we can do that. So the goal of 6.01 is, then, really to convey a distinct perspective about how we engineer systems.

Now, having said that, this is not a philosophy course. We are not going to make lists of things to do if you want it to be robust. We're going to learn to do things by actually making systems. This is an introductory engineering course. And so you're going to build things.

The idea is going to be that in constructing those things, we've written the exercises so that some of those important themes become transparent. So the idea is -- this is introductory engineering. You'll all make things. You'll all get things to work, and in the process of doing that, learn something about the bigger view of how quality engineering happens.

So despite the fact that we're really about modes of reasoning, that will be grounded in content. We selected the content very broadly from across EECS. EECS is an enormous endeavor. We can't possibly introduce everything about EECS in one subject. That's ridiculous.

However, we wanted to give you a variety. We wanted to give you a sense of the variety of tasks that you can use, that you can apply the same techniques to. So we want to introduce modes of reasoning, and then show you explicitly how you can use those modes of reasoning in a variety of contexts. So we've chosen four, and we've organized the course around four modules. First module is software engineering, then signals and systems, then circuits, then probability and planning.

Even so, even having chosen just four out of the vast number of things we could have chosen, there's no way we can tell you adequately-- we can't give you an adequate introduction to any of those things either. What we've chosen to do instead is focus on key concepts represented by the asterisks. The idea is going to be we choose one or two things and really focus on those deeply so you get a thorough understanding not only of how that fits within, for example, the context of software engineering, but also how that concept ramifies into other areas.

Notice that I tried to choose the stars so they hit multiple circles. That's what we're trying to do. We're trying to not only introduce an idea to you, but also show you how it connects to other ideas. So the idea, then, is to focus on a few, we hope, very well-chosen applications that will demonstrate a variety of powerful techniques.

Our mantra, the way we intend to go about teaching this stuff, is practice, theory, practice. There's an enormous educational literature that says-- whether you like it or not-- people learn better when they're doing things. You have a lot of experience with that. You have a lot of experience on the other side, too. I'll try to forget the other side, or at least try to wipe it from your brain momentarily to focus on your more fundamental modes of learning.

When you were a kid and you were learning your first language, you didn't learn all the rules of grammar first. You didn't learn all the letters of the alphabet first. You didn't learn about conjugating verbs first. You learned a little bit about language. You started to use it. You ran into problems. You learned a little more about language. You learned to go from words like "feed me" to higher level concepts, like "Hey, what's for dinner?"

So the idea is that you learned it in an iterative process where you learned some stuff, tried it out, learned some more stuff, tried it out. And it built up. There's an enormous literature in education that says that's exactly how we always learn everything.

And so that's the way this course is focused. What we will do is, for example, for today, we'll learn a little bit about software engineering. Then, we'll do two lab sessions where you actually try to use the things we talk about. Then, we'll come back to lecture and we'll have some more theory about how you would do programming. And then, you go back to the lab and do some more stuff. And the hope is that by this tangible context, you'll have a deeper appreciation of the ideas that we're trying to convey.

So let me tell you a little bit about the four modules that we've chosen. The course is going to be organized on four modules. Each module will take about one fourth of the course.

First thing we'll look at is software engineering. As I said, we don't have time to focus on, or even survey, all of the big ideas in software engineering. It's far too big. So we're going to focus narrowly on one or two things.

We'd like you to know about abstraction and modularity because that's such an important idea in the construction of big systems. So that's going to be our focus. In today's lecture, we'll begin talking about modularity and abstraction at the small scale. How does it affect the things you type as instructions to a computer?

But by next week, we're going to be talking about a whole bigger scale. By next week, we're going to talk about constructing software modules at a much higher level. In particular, we'll talk about something that we'll call a state machine.

A state machine is a thing that works in steps. On every step, the state machine gets a new input. Then, based on that input and its memory of what's come before, the state machine decides to do something. It generates an output. And then, the process repeats.

We will see that that kind of an abstraction -- state machines -- there's a way to think about state machines that is compositional that you can think of as a hierarchy, just as you can think of low-level hierarchies within a language. I'll say a lot more about that today.

So the idea will be that once you've composed a state machine, you'll be able to join two state machines and have its behavior look just like one state machine. That's a way to get a more complicated behavior by constructing two simpler behaviors. That's what we want. We want to learn tools that let us compose complex behaviors out of simple behaviors. And the tangible model of that will be the robot. We will see how to write a program that controls a robot as a state machine.

That's certainly not the only way you could control a robot. And it's probably not the way you would first think of it if you took one course in programming and somebody said to you, go program the robot to do something. What we will see is that it's a very powerful way to think about it for exactly this reason of modularity.

The bigger point that we will make in thinking about this first module is the idea of, how do you make systems modular? How do you use abstraction to simplify the design task? And in particular, we will focus on something that we'll call PCAP.

When you think about a system, we will always think about it in terms of, what are the primitives? How do you combine them? How do you abstract a bigger behavior from those smaller behaviors? And what are the patterns that are important to capture?

So the bigger point is this idea of PCAP, which we will then revisit in every subsequent module.

OK, second module is on signals and systems. That's also an enormous area. So we only have time to do one thing. The thing that we will do is we will think about discrete time feedback. How do you make a system that's cognizant of what it's done so that it, in the future, can do things with awareness of how it got there?

A good example is robotic steering. So the idea is going to be, OK, think about what you do when you're driving a car. And think about how you would tell a robot to do that same thing. Here's a naive driving algorithm. I don't recommend it, but it's widely used in Boston, apparently.

[LAUGHTER]

I find myself to the right of where I would like to be. So what should I do? Turn left. I'm still to the right of where I'd like to be. What should I do? Turn left. Oh! I'm exactly where I should be. What should I do? Go straight ahead. Oh, that's a bad idea.

And what we'll see is that perfectly innocent looking algorithms can have horrendous performance. What we'll do is try to make an abstraction of that. We'll try to make a model. We'll try to capture that in math so that we don't need to build it to see the bad behavior.

We'll make a model. We'll use the model to predict that that algorithm stinks. But more importantly, we'll use the model to figure out an algorithm that'll work better. In fact, we'll even be able to come up with bounds on how well such a controller could possibly work.

So the focus in this module is going to be, how do you make a model to predict behavior? How do you analyze the model so that you can design a better system? And then, how do you use the model and the analysis to make a well-behaved system?

The third module is on circuits. Again, circuits is huge. We don't have time to talk about all of circuits. We'll do very simple things.

We'll focus our attention on how you would add a sensory capability to an already complicated system. The idea is going to be to start with a robot-- I guess this is brighter-- start with our robots and design a head for the robot.

The robot comes from the factory with sonar sensors. The sonar sensors are these things. There's eight of them. They tell you how far away something that reflects the ultrasonic wave is.

As they come from the factory, the robots can't sense light. What you'll do is add light sensors. The goal is to make a system to modify the robot so that the robot tracks light. That's a very simple goal.

And the way we'll that is to augment the robot with a simple sensor here, showed a little more magnified here. The idea is that this is a LEGO motor. The LEGO motor will turn this relative to the attachment. That's the robot head's neck. So the robot will be able to do this.

And the robot will have eyes. These are photosensors, photoresistors, actually. So the idea is going to be that there's information available in those sensors for figuring out where light is so that you can track it.

Your job will be to build a circuit-- that that's this thing-- that connects via cables-- these red cables and yellow cables-- connects via cables over to this head. We'll give you the head. Your job will be to make the circuit that converts the signal from the photoresistor-- which is in proportion to light-- and figures out how to turn the motor to get the head to face the light and then ship that information down to the robot to let the robot turn its wheels to get the body. So it's kind of like the light comes on bright over here. The robot looks at it and says, oh, yeah, that's where I want to be. So that's the idea in the third module is to incorporate new sensing capabilities into the robot.

The final module is on probability and planning. And the idea there is to learn about how you make systems that are robust to uncertainty and that can implement complicated plans, that they, too, are robust to uncertainty.

So there's a number of things that we will do, including creating maps of spaces that the robot doesn't understand, telling the robot how to localize itself, how if it woke up suddenly in an environment, it could figure out where it is, how to make a plan. And as an example, I'll show you the kind of system that we will construct.

Here, the idea is that we have a robot. The robot knows where it is. Imagine there's a GPS in it. There isn't, but imagine there is. So the robot knows where it is, and it knows where it wants to go. That's the star.

But it has no idea what kind of obstacles are in the way. So if you were a robotic driver in Boston, you know that you started out at home and you want to end up in MIT. But there's these annoying obstacles, they're called people, that you should, in principle at least, miss. So that's kind of the idea.

So I know where I am. I'm the robot. I know where I am. I know where I want to be. And I'm going to summarize that information here. Where I am is purple. Where I want to be is gold.

And I have a plan. That's blue. My plan's very simple. I don't know anything about anything other than I'm in Waltham and I want to go to Cambridge. So blast east. So I imagine that the best way to do there is a straight line.

OK, so now what I'm going to do is turn on the robot. The robot has now made one step. And I told you before about these sonar sensors. From the sonar sensors, the robot has learned now that there seems to be something reflecting at each of these black dots. It got a reflection from the black dots, from the sonar sensors. That means there's probably a wall there, or a person, or something that, in principle, I should avoid.

And the red dots represent, OK, the obstacle is so close I really can't get there. So I'm excluded from the red spots because I'm too big. The black spots seem to be an obstacle. The red spots seem to be where I can't fit. I still want to go from the where I am, purple, to where I want to be, gold.

So what I do is I compute the new plan. OK then, I start to take a step along that plan. And as I'm stepping along, OK, so now, I think that I can't go from where I started over to here. I have to go around this wall that I didn't know about initially.

So now I just start driving. And it looks fine, right? I'm getting there, right? Now, I know I can go straight down here.

Oh, wait a minute. There's another wall. OK, what do I do now? So as the robot goes along, it didn't know when it started what kinds of obstacles it would encounter. But as it's driving, it learned. Oh, that didn't work. Start over!

So the idea is that this robot is executing a very complicated plan. The plan has, in fact, many sub-plans. And the sub-plans all involve uncertainty. It didn't know where the walls were when it started. And when it's all done, it's going to have figured out where the walls were and-- provided there's a way-- presumably find the way to negotiate the maze and get to the destination.

So the idea, then, is that if you were asked to write a conventional kind of program for solving that, it might be kind of hard because of the number of contingencies involved. What we will do is break down the problem and figure out simple and elegant ways to deal not only with uncertainty, but how do you make complex plans.

So as I said, our primary pedagogy is going to be practice, theory, practice. And so that ramifies in how the course is organized. So this is a quick map of some of the aspects of the course.

So we'll have weekly lectures. It's lecture unintensive. In total, there's only 13 lectures. We'll meet once a week here for lecture.

There's readings. There's voluminous readings. There's readings about every topic that we will talk about. And the readings were specifically designed for this course. I highly recommend that you become familiar with the readings. If you have a question after lecture, it's probably there. It's probably explained.

We will do online tutor problems. We sent you an email if you pre-registered for the course. So you may already know about this. The idea is going to be that there's ways that you can prepare for the course by doing computer exercises. And we will also use those same kinds of exercises in all of the class sessions.

We will have two kinds of lab experiences. Besides lecture, the other two events that you have to attend are a software lab and a design lab. That's the practice part. So after you learned a little bit about the theory by going to lecture, by doing the reading, then you go to the lab and try some things out.

We call the first lab a software lab. It's a short lab. It's an hour and a half. You work individually. You try things out. You write little programs. The courseware can check the program to see if it's OK. And primarily, the exercises in the software lab are due during the software lab. But on occasion, there will be extra things due a day or two later. The due dates are very clearly written in the tutor exercises.

Once a week, there's a design lab. That's a three hour session in which you work with a partner. The reason for the partner is that the intent-- the difference between the design labs and the software labs is that the design labs ask you to solve slightly more open-ended questions, the kind of question that you might have no clue what we're asking. Open-ended, the kind of thing that you will be asked to do after you graduate. Design the system. What do you mean, design the system?

So the idea is that working with a partner will give you a second, immediate source of help and a little more confidence if neither of you knows the solution so that you raise your hand and say, I don't have a clue what's going on here. So the idea is that once a week we do a software lab individually. Once a week, we do a design lab, a little more open-ended with partners.

There's a little bit of written homework, four total. It's not much compared to other subjects. It's mostly practice.

There's a nano-quiz, just to help you keep pace to make sure that you don't get too far behind. The first 15 minutes of every design lab starts with a nano-quiz. The nano-quizzes are intended to be simple if you've caught up, if you're up to date. So the idea is that you go to design lab, the first thing you do is a little, 15 minute nano-quiz. The nano-quiz uses a tutor much like the homework tutor, much like the Python tutor. And it's intended to be simple.

But it does mean please get to the design lab on time. The nano-quizzes are administered by the software. It starts the hour when the design lab starts. It times out 15 minutes later. So if you come 10 minutes late, you will have 5 minutes to do something that we planned to give you 15 minutes for.

We will also have exams and interviews. The interviews are intended to be a one-on-one conversation about how the labs went. And we will have two mid-terms and a final.

So that's kind of the logistics. The idea behind the logistics is practice, theory, practice. Come to the labs. Try things out. Make sure you understand. Develop a little code. Type it in. See if it works. If it works, you're on top of things. You're ready to get the next batch of information from the lecture and readings.

OK, let's go on, and let's talk about the technical material in the first module of the course, in the software module. We kick the course off talking about software engineering for two reasons. We'd like you to know about software engineering. It's an incredibly important part of our department. It's an incredibly important part of the engineering of absolutely any system, any modern system.

But we'd also like you to know about it because it provides a very convenient way to think about-- it's a convenient language to think about the design issues, the engineering issues in all the other parts of the class. So it's a very good place to start.

So what I will do today is talk about some of the very simplest ideas about abstraction and modularity in what I think of as the lowest level of granularity. How do you think about abstraction and modularity at the micro scale, at the individual lines of code scale? As I said earlier, we will, as we progress, look at modularity and abstraction at the higher scale. But we have to start somewhere. And we're going to start by thinking about, how do you think about abstraction and modularity at the micro scale?

Special note about programming. So what we are trying to do is, in the first two weeks, ramp everybody up to some level of software security, where you feel comfortable. So the first two weeks of this course is intended to make you comfortable with programming. We don't assume you've done extensive programming before. We want you to become comfortable that you're not behind. And that's the focus of the first two weeks' exercises.

If you have little or no previous background, if you are uncomfortable, please do the Python tutor exercises. If you have not -- if you do not have a lot of experience programming, if you're uncomfortable with the expectation that you can do programming, do that first. That takes priority over all the other assignments during the first two weeks.

In particular, if you're uncomfortable, we will run a special Python help session on Sunday. And if you attend that, you can get a free extension. The idea is completing the tutor exercises is intended to make you feel comfortable that you have the software background to finish the rest of the course. Do that first. We will forgive falling behind in other things so that you feel comfortable with programming.

If, at the end of two weeks, you still feel uncomfortable, we have a deal with 6.00, the Python programming class, that they will allow you to switch your registration from 6.01 to 6.00. But that expires Valentine's Day.

[LAUGHTER]

So you have to make up your mind before Valentine's Day if you'd like to use that option. So the idea is we'd like you to be comfortable with programming. If you haven't programmed before, do the Python tutor exercises. Go to software lab. Go to design lab, but work on the tutor exercises. The staff will help you with them. You can go to office hours. There's office hours listed on the home page. You should try to become comfortable, and you should try to set as your goal -- I'm going to be comfortable before Valentine's Day. And if you're not, talk to a staff member about that.

OK, so what do I want you to know about programming? Well, we're going to use

Python. We selected Python because it's very simple and because it lets us illustrate some very important ideas in software engineering in a very simple context. That's the reason.

One of the reasons that it's simple is that it's an interpreter. After some initialization, the behavior of Python is to fall into an interpreter loop. The interpreter loop is, ask the user what he would like me to do, read what the user types, figure out what they're talking about, and print the result, repeat-- very simple.

What that means is that you can learn by doing. That's one of the points of today's software lab. You can simply walk up to a computer, type the word python-- what you type is in red. Type the word "python." It will prompt you, so this chevron, that says, I'd like you to tell me something to do. I have nothing to do.

If you type "2," Python tries to interpret that. In this particular case, Python says, oh, I see. That's a primitive data item. That's an integer. This person wants me to understand an integer. And so it will echo 2, indicating that it thinks you want it to understand a simple integer.

Similarly, if you type 5.7, it says, oh, I got that. That's a float. The person wants me to remember a floating point number. And it will similarly echo the float.

Now, of course, there's no exact representation for floats, right? There's too many of them, right? There's a lot of them. There's even more floats than there are ints, right? So it has an approximation. So it will print its approximation to the float that it thinks you are interested in.

If you type a string, "Hello," it'll say, oh, primitive data structure, string. And it'll print out that string. So the idea is one of the features of Python that makes it easy to learn is the fact that it's interpreter based. You can play around. You can learn by doing.

Now, of course, if the only thing it did was simple data structures, it would not be very useful. So the next more complex thing that it can do is think about combinations. If you type "2 + 3," it says, oh, I got it. This person's interested in a combination. I should combine by the plus operator two ints, 2 and 3. Oh, and if I do that, if I combine by the plus operator two and three, I'll get 5. So it prints 5. So that's a way you know that it interprets "2 + 3" as 5.

Similarly here, except I've mixed types. "5.7 + 3," it says, oh, this user wants me to apply the plus operator on a float and an int. OK, well I'll upgrade the int to a float. I'll do the float version, and I'll get this, which is its representation of 8.7.

So the idea is that it will first try to interpret what you're saying as a simple data type. If that works, it prints the result to tell you what it thinks is going on. It then will try to interpret it as an expression. And sometimes, the expressions won't makes sense. In particular, if you try to add an int to a string, it's going to say, huh?

And over the course of the first two weeks, we hope that you get familiar with interpreting this kind of mess. That's Python's attempt to tell you what it was trying to do on your behalf and can't figure out what you're talking about.

OK, so that was simple. But it already illustrates something that's very important, and that's the idea of a composition. So the way Python works, the fact that when you added 3 to 2 it came out 5, what we were doing was composing complicated-- well, potentially complicated (that was pretty simple) -- potentially complicated expressions and reducing them to a single data structure.

And so that means that, in some sense, this operation, 3 times 8, can be thought of as exactly the same as if the user had typed in 24. Whenever you can substitute for a complex expression a simpler thing, we say that the system is compositional. That's a very powerful idea.

Even though it's simple, it's a very powerful idea. And it's an idea that you all know. You've seen it before in algebra, in arithmetic. So in arithmetic expressions, you can think about how the sum of two integers is an int. That's a closure. That's a kind of a combination that makes the system compositional and that provides a layer of hierarchical thinking so that, in your head, even though it says 3 times 8, you don't need to remember that anymore. You can say, oh, for any purposes that follow, I might just as well think of 3 times 8 as being a single integer, 24.

It's part of many other kinds of systems, for example, natural language. The simplest example in natural language is that you can think about "Apples are good as snacks". "Apples" is a noun. It's a plural noun. Or you could substitute "Apples and oranges", and it makes complete sense within that same structure.

So "Apples and oranges are good as snacks". The combination of "apples" and "oranges" works in every way from the point of view of the grammar in the same way that a simple noun, "apples," worked. What we would like to do is use that idea as the starting point for a more general compositional system.

And a good way to think about that is by way of names. What if we had some sequence of operations that we think is particularly important so that we would like to somehow canonize that so that, subsequently, we can use that sequence of operations easily?

Python provides a very simple way to do it. Every programming language does. It's not unique to Python. But the idea is -- so here's an example. "2 times 2" -- I'm squaring 2 and get 4. "3 times 3" -- I'm squaring 3, and I'm getting 9. "8 plus 4 times 8 plus 4", I'm squaring "8 plus 4". "8 plus 4", well, I can think of that as 12. I'm squaring 12, I'm getting 144.

The thing I'm trying to illustrate there is the notion of squaring. Squaring is a sequence of operations that I would like to be able to canonize as a single entity so that, in subsequent programs, I can think of the squaring operation as a single operation just like I think of times. The way we say that in Python is "define square of x to be return x squared". Then, having made that definition, I can say "square of 6", and the answer is 36.

OK, this is a very small step. But it illustrates a very important point, the idea being that Python provides a compositional facility. And it's hierarchical. Having defined square, I can use square just as though it were a primitive operator. And I can use square to define higher level operations.

So for example, what if I were interested in doing lots of sums of squares? Say I'm Pythagoreas, right? So I might want to add the square of 2 and the square of 4 to get 20, or the square of 3 with the square of 4 to get 25. Using that simple idea of composition, we can write a new program, sumOfSquares. sumOfSquares takes two arguments, x and y. And it returns the square of x and the square of y.

SumOfSquares doesn't care about how you compute the square. It trusts that square knows how to do that. So the work is smaller. The idea is that square takes care of squaring single numbers. sumOfSquares doesn't have to know how to square numbers. It just needs to know how to make a sum of squares.

So what we've done is we've broken a task, which was not very complicated, but the whole idea is hierarchical. We've taken a problem and broken it into two pieces. We factored the problem into how do you do a square, and how do you sum squares. And the idea, then, is that this hierarchical structure is a way of building complex systems out of simpler parts. So that's the idea of how you would build programs that are compositional.

Python also provides a utility for making lists, for making data structures that are compositional. The most primitive is a list. So in Python, you can specify a list. Here's a list of integers. So the list says, beginning list, end of list, elements of list. So there's five elements in the list, the integers 1, 2, 3, 4, 5.

Python doesn't care what the elements of a list are. We'll see in a minute that that's really important. But for the time being, the simplest thing that you can imagine is a heterogeneous list. It's not critical that the list contain just integers. Here's a list that contains an int, a string, an int, and a string. Python doesn't care. It's a list that has four elements. The first element's an int. The second element's a string, et cetera.

Here's an even more complex example. Here's a list of lists. How many elements are in that list? Three. How many elements are in that list?

So the idea is that you can build more complex data structures out of simple ones. That's the idea of compositional factoring applied to data. Just like it was important when we were thinking about procedures, to associate names with procedures-- that's what "def" did-- we can also think about associating names with data structures. And that's what we use something that Python calls a variable for.

So I can say "b is 3". And that associates the data item, 3, with the label, b. I can say, "x is 5 times 2.2". Python will figure out what I mean by the expression on the right. It'll figure out that I'm composing by using the star operator, which is multiply, an integer and a float, which will give me a float. The answer to that's going to be a floating point number. And it will assign a label, x, to that floating point number.

You can have a more complicated list, a data structure, and associate the name y with it. Then, having associated the name y, you get many of the same benefits of associating a name with a data structure that we got previously in associating a name with an operation. So we can say, y(0). And what that means is, what's the zero-th elements of the data structure, y?

So the zero-th element of the data structure, y, is a list, [1, 2, 3]. Python has some funky notations. The -1 element is the last one. So the -1th element of y is [7, 8, 9]. And it's completely hierarchical.

If I asked for the -1 element of y, I get [7, 8, 9]. But then, if I asked for the first element of that result, I get 8. OK? Everything is clear?

OK, just to make sure everything is clear, I want to ask you a question. But to kick off the idea of working together, I'd like you to think about this question with your neighbor. So before thinking about this question, everybody stand up. Introduce yourself to your neighbor.

[AUDIENCE TALKS]

So now, I'd like you to each discuss with your neighbor the list that is best represented by which of the following figures, 1, 2, 3, 4, or 5, or none of the above. And in 30 seconds, I'm going to ask everybody to raise a hand with a number of fingers indicating the right answer. You're allowed to talk. That's the whole point of having a partner.

[AUDIENCE TALKS]

OK. I'd like everybody now to raise their hand. Put up the number of fingers that show the answer. And I want to tally. Fantastic! Everybody gets it.

OK, so which one do you like?

AUDIENCE: 3.

PROFESSOR: 3 -- why do you like three. Somebody explain this to me? It just looks good? Its pattern recognition. What's good about 3?

AUDIENCE: It shows the compositional element of the list.

PROFESSOR: Compositional? What is the compositional element in the pictures? What represents what? OK, 'a' represents a. That's pretty easy, right? So that takes care of the bulk of the figures.

What's the blue lines represent? Someone else? I didn't quite understand.

AUDIENCE: The angles represent like a list.

PROFESSOR: They represent a list. Where is the list on the figures?

AUDIENCE: The vertex?

PROFESSOR: The vertex. The vertices are lists. So in 3 -- at the highest level, we have a list that's composed of how many elements? 2. The first element of that list is?

AUDIENCE: a.

PROFESSOR: And the second element of that list is?

AUDIENCE: Another list.

PROFESSOR: Another list. That's the hierarchical part, right? That second list has how many elements?

AUDIENCE: 2.

PROFESSOR: Fine, good, recurse. You got it. What is the list represented by number 2? A single list with five elements. Square bracket, a, comma, b, comma, c, comma, d, comma, e, square bracket, right? What is the list represented by that one?

AUDIENCE: Not a list.

PROFESSOR: Agh! It's not a list! What is it? Who knows?

AUDIENCE: Looking at the variable names, it defines them. You have variables. You have a variable a, that defines a list that contains b, and the variable, c, that defines another list that contains d.

PROFESSOR: So we could make that a variable. If we said a is a variable that comprises b and c, then we have the problem of how we're going to associate variables and elements into this list, right? So the weird thing about this one and, let's see, that one's weird. This one's also kind of weird. This one's weird because we're giving names to lists in a fashion that's not showed up here, right? That's not to say you couldn't invent a meaning. It's just that it doesn't map very well to that representation.

Similarly over here, we seem to be giving the name b to the element a, and then the name c to the element b. What on earth are you talking about? It's not clear what we're doing their either. So the point is to get you thinking about the abstract representation of lists and how that maps into a complex data structure. That was the whole point.

OK, so we've talked about, then, four things so far. How do you think about operations in a hierarchical fashion. And the idea was composition. We think about composing simple operations to make bigger, compound operations. That's a way of saying, there's this set of operations that I want to call foo. So every time I do this complicated thing that has three pages of code, that's one foo. And that's a way that we can then combined foos in some other horribly complicated way to make big foos. So the idea is composition. That's the first idea.

The second is associating a name with that composition. That's what "def" does-- define name, name of a sub-routine. So we thought about composing operations, associating names with them. We composed data in terms of lists, and we associated names with those lists in terms of variables.

The next thing we want to think about is a higher order construct where we would like to conglomerate into one data structure both data and procedures. Python has a concept called a class that lets us do that. In Python, you make a new class by saying to the Python prompt, I want a new class called Student. And then, under Student, there is this thing which we will call an attribute. An attribute to a class is simply a data item associated with the class. And a method-- a method is just a procedure that is associated with the class.

So there's this single item class called Student that has one piece of data, its attribute, school, and one procedure, which is the method calculateFinalGrade. So then, this is the kind of data structure you might imagine that a registrar would have. It's a way to associate.

So the idea here is that everybody here is a student. They all have a school. And they all have a way of calculating their final grade. That's a very narrow view that maybe a registrar would have.

So classes, having defined them, we can then use the class to define an instance. So an instance is a data structure that inherits all of the structure from the class but also provides a mechanism for having specific data associated with the instance.

So in Python, I say Mary is a student. By mentioning the name of the class and putting parenthesis on it, I say, give me an instance of the student. So now, Mary is a name associated with an instance of the class, Student. John is similarly an instance of the class, Student. So both Mary and John have schools. In fact, they're both the same. The school of Mary and the school of John are both MIT.

But I can extend the instance of Mary to include a new attribute, the section number, so that Mary's section number is 3 and John's section number is 4. So this provides a way-- it's a higher-order concept. We thought of a way to aggregate operations into complicated operation, data into complicated data. Classes aggregate data and operations. Classes allow us to create a structure and then generate instances. And then the instances have access to those features that were defined in the class, but also have the ability to define their own unique attributes and methods.

You can also use a class to define a subclass. So here, I'm defining the subclass, Student601. All Student601s are members of the class, Student. The reverse is not true. So all Student601 entities inherit everything that a Student has. But all 601 students share some other things. Besides having a school which all students have, 601 students also have a lecture day, a lecture time, and a method for calculating tutor scores. Not all students have a method for calculating tutor scores. But members of the class Student601 do. So this, again, represents a way of organizing data and operations in a way that makes it easier to compose higher, bigger, more complex structures.

The final thing that I want to talk about today is the specific, gory details for how Python manages the association between names and entities. We've already seen two of those. Naming operations is via "def." And it gives rise to the name of a procedure. Variables are ways of naming data structures. Now, we've seen a way of naming classes. And in fact, it's helpful if you understand.

So Python associates names and entities in a very simple, straightforward fashion. And if you know the ground rules, it makes it very easy to deal with. And if you don't know the ground rules, it makes it very hard to deal with. So what's the ground rules? Here's the gory details.

So Python associates names with values in what Python calls a binding environment. An environment is just a list that associates a name and an entity. So if you were to type b equals 3 what Python is actually doing is it's building this environment. When you type b equals 3, it adds to the environment a name, b, and associates that name with the integer, 3.

When you type x equals 2.2, it adds a name, x, and associates it with the float, 2.2. When you say foo is minus 506 times 2, it makes the name, foo, and associates it with an int, minus 1012.

Then, if you ask Python about b, the rule is look it up in the environment and type the thing that b refers to. So when you type "b," what Python really does is it goes to the environment. It says, do I have some entity called "b?" Well, yes I do. It happens to be an int, 3. So it prints 3.

If you ask, what is "a?" Python says, OK, in my environment, do I have some name, "a?" It doesn't find it. So it prints out this cryptic message that basically says, sorry, guys, I can't find something called "a" in the current environment.

That's the key to the way Python does all name bindings. So in general, there's a global environment. You start typing to Python. It just starts adding and modifying the bindings in the binding environment. So if you type a equals 3 and then type "a," it'll find 3. If you then type "b=a+2," it evaluates the right-hand side relative to the current environment.

So it first looks here. And it says, do I have something called "a?" Ah, yes. It's an integer, 3. Substitute that. Do I know what 2 is? Oh yeah, that's just an int. Do I know what plus is? Oh yeah, that's the thing that combines two ints. So it decides that a plus 2-- it evaluates a plus 2 in the current environment. It gets 5.

And it says, oh, I'm trying to do a new equals, a new association, a new variable. Make the name, b, points to this evaluated in the current environment. So b gets associated with int 5. Then, if I do this line, it evaluates b plus 1 in the current environment. b is 5 in the current environment. It adds 1. It gets 6. And then, it says, associate this thing, 6, with b. So it overwrites the b, which had been bound to 5, and b is now bound to 6. OK?

So the whole thing, the way it treats variables, the way Python associates a name with a value in a variable, is evaluate the right-hand side according to the current environment. Then, change the current environment to reflect the new binding.

What it does in the case of sub-routines is very similar. So here's an illustration of the local environment that is generated by this piece of code.

When I say a equals 2, it generates a name in the local environment, a. It evaluates the right-hand side and finds 2. So it makes a binding in the local environment where the name, a, is associated with the integer, 2.

Then, I say define square of x to be return x squared. That's more complicated. Python says, oh, I'm defining a new operation. It's a procedure. The procedure has a formal argument, x. It has a body, return x times x. I'm going to have to remember all of that stuff.

So I'm trying to define a new procedure called square. It's going to make a binding for square. So in the future, if somebody says the word square, it'll find out, oh, square I remember that one. square, it's a procedure. Just like the binding for a variable might be an int, the binding for a procedure is the name of the procedure.

Then, in the procedure, which is some other data structure outside the environment, it's got to remember the formal parameters-- in this case, x-- and the body. And for the purpose of resolving what do the variables mean, it needs to remember what was the binding environment in which this sub-routine was defined. So that's this arrow.

So this sequence says, make a new binding square, points to a procedure. The procedure has the formal argument, x. It has the body return x times x. And it has the binding. It came from the environment, E1, the current environment.

OK, is everybody clear? So the idea is that the environment associates names with things. The thing could be a data item, or it could be a procedure. Then, when you call a procedure, it makes a new environment. So what happens, then, when I try to evaluate a form, square of a plus 2?

What Python does is it says, OK, I need to figure out what square is. So it looks it up in the environment, and it finds out that square is a procedure. Fine, I know how to deal with procedures. So then, it figures out this procedure has a formal argument,

x. Oh, OK, if I'm going to run this procedure, I'm going to have to know what x means.

So Python makes a new environment-- here, it's labelled E2, separate from the global environment, E1. It makes a new environment that will associate x with something. Doesn't know what it is yet, it just knows that this square is a procedure that takes a formal argument, x. So Python makes a new environment, E2, with x pointing to something.

Then, Python evaluates the argument a plus 2 in the environment E1. You called square of a plus 2 in the environment of E1. So it figures out what did you mean by a plus 3. Well, you were in the environment E1. So it means whatever a plus 3 would have meant if he had just typed a plus 3 in that environment. So you evaluate a plus 3 in the environment E1, and you get 5.

So then, this new environment, E2, that is set up for this procedure, square, associates 5 with x. Now it's ready to run the body. So now, it runs this procedure, return x times x. But now, what it's trying to resolve its variables, it looks it up in E2. So it says, I want to do the procedure, the body, x times x. I need to know what x is, and I need to know it twice. Look up what x means, but I will look it up in my E2 environment that was built specifically for this procedure. And fortunately, there's an x there. So it finds out that x is 5. It multiplies 5 times 5. It gets the answer is 25. It returns 25. And then, it destroys this environment, E2, because it was only necessary for the time when it was running the procedure body. Is that clear?

OK, so a slightly more difficult example illustrates what happens whenever everything is not defined in the current local environment. What if I type define biz of a? Well, I create a new name in the local environment that points to a procedure. The procedure has a formal parameter, a, and a body that returns a plus b. The procedure also was defined within the environment E1, which I'll keep track of.

Then, if I say b equals 6, that makes a new binding in the global environment, b equals 6. Then, if I try to run biz of 2, look up biz. Oh, that's a procedure, formal parameter, a. Make an environment, has an a in it. What should I put in a? Evaluate the argument, 2. OK, a is 2. Put two here.

Now, I'm ready to run the body. Run the body in the environment, E2. When I run return a plus b in E2, I have to first figure out a. Well, that's easy. a is 2. Then, I have to figure out b. What's b?

AUDIENCE: 6?

PROFESSOR: 6. So how did you get 6?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So this local environment that was created for the formal parameter has, as its parent, E1 because that's where the procedure was defined. So it doesn't find b in this local environment. So it goes to the parent. Do you have a "b?" And it could, in principal, propagate up a chain of environments. So you could construct this hierarchically. So it will resolve bindings in the most recent environment that has that binding. So the answer, then, is that when you run biz of 2, this b gets associated with that b, OK?

So that's how the environments work for simple procedures and simple data structures. It's very similar for the way it works with classes. So imagine that I had this data, and I wanted to represent that in Python. What I might do is look at the common features. The courses are all the same. The rooms are all the same. The buildings are all the same. The ages are highly variable.

So I might want to create a class that has the common data. So I might do this-- class Staff601. The course is 601. The building's 34. The room is this.

The way Python implements a class is as an environment. Executing this set of statements builds the class environment. This is it. It's a list of bindings. Here, I'm binding the name, course, to the string, 601, et cetera. If there were a method, I would do the same thing, except it would look like a procedure then.

So this creates the Staff601 environment. Staff601, because I executed this class statement, that creates a binding in the local environment, Staff601, which points to the new environment. So now, in the future, when Python encounters the name Staff601, it will discover that that's an environment. Python implements classes as environments.

So now, when I want to access elements within a class, I use a special notation. It's a dot notation. Python regards dots as ways of navigating an environment. When Python parses staff.room, it looks up Staff601 in the current environment. If it finds an environment, it then says, oh, I know about this .room thing. All I do is I look up the room name in the environment Staff601. And when it does that, it gets the answer 501.

And the same sort of thing happens here. It looks up Staff601. It finds an environment. It looks up coolness. It finds out there is no such thing. Well, no, that's not true. So it creates coolness within 601 and assigns an integer, 11, to it.

So then, the way Python treats methods is completely analogous-- oh, excuse me, instances. I'm doing instances first. If I make pat be an instance of Staff601, pat is an instance of the class Staff601. pat is implemented as an environment.

So when I make pat, pat points to a new environment-- here, E3. The parent of E3 is the class that pat belongs to, which is, here, E2. And when I make the instance, it's empty. But now, if I ask what is pat.course, well, pat points to this environment. Does this environment have something called a course? No. Does the parent? Yes. Course is bound to the string 601. So pat.course is 601 just the same as Staff601.course had been 601.

pat is an instance. It's a new environment with the class environment as its parent. You can add attributes to instances. And all that does is populate the environment associated with the instance. You can add methods to classes. And that does the same thing.

So here, I've got the class, Staff601, which has a method, salutation, instance variables, course, building, and room. So when I build that structure, Staff601 points to an environment that contains salutation, which is a procedure, in addition to a bunch of instance variables.

So now, all of the rules that we've talked about with regard to environments apply now to this class. So in particular, I can say Staff601 salutation of pat. When Python parses Staff601, it finds an environment. It says dot salutation. Oh, I know how to do that. Within the environment, Staff601, look for a binding for the name salutation. Do I find one? Well, yeah, there it is. It points to a procedure. So staff dot salutation is a procedure. Do just the same things that we would have done with a simple procedure.

The only difference here is that the procedure came from a class. In this particular case, the sub-routine that I define has a formal parameter, self. So then, that's going to have to build when I try to evaluate it. That has to build a binding for self, which is set to pat. pat was an environment. So self gets pointed to pat. So now, when I run Staff601.salutation on pat, it behaves as though that generic method was applied to the instance pat.

We'll do that a lot. It's a little bit of redundancy. We know that pat is a member of Staff601. So we will define a special form-- or I should say, Python defines a special form-- that makes that easy to say. This is the way we will usually say, the instance pat should run the class method salutation on itself. This is simply a simplified notation that means precisely that, OK?

So what we covered today, then, was supposed to be the most elementary ideas in how you construct modular programs, Modularity at the small scale. How do you make operations that are hierarchical, data structures, and classes? What we will do for the rest of the week is practice those activities.