PROFESSOR: Hello. Welcome to the second lecture in 6.01. I hope you had a good time last week. Last lecture, we looked at what is probably the most important theme in this course, which is, how do you organize your thoughts, your design, the things that you do, in order to manage complexity when you're trying to build a complicated system?

The mantra for this class is PCAP-- primitives, combinations, abstractions, and patterns. And last time, we saw how at the very most elementary level, Python provides some tools with which we can achieve this goal of defining primitives, combining them into more complicated ideas, abstracting the important information, and generating and capturing new patterns.

And so, for example, we saw that Python has a defined statement that lets you associate a sequence of operations with a name-- both of those things are important, the sequence represents a combination, the name represents a way that we can abstract the behavior of the combination and treat it as though it were primitive operation.

We saw that we could do the same sort of thing for data structures. And in particular, the list structure in Python allows us to generate hierarchical heterogeneous structures in much the same way. Then variables allow us to associate names with those structures. And finally, we saw that classes allow us to combine not only data, but also procedures -- all into one object of related things.

So that's PCAP at the most primitive level. What I want to do today is talk about PCAP at higher levels. How do you build upon that framework to continue this idea of building with abstraction and modularity? How do you make-- how do you combine primitive operations into powerful operations? How do you combine

primitive data into powerful data?

And what I want to do is think about the next level above the most rudimentary level in Python. So we'll look first at some programming styles, and how that affects your ability to define useful abstractions. And then I'll look at something much higher level, which is state machines, which is the way that we will think about the construction of robot controls.

So I'll start with just a few words about how you-- the different ways that you could structure a program. The reason for doing this is that the basic structure that you use can have an important effect on your ability to abstract. We'll look at three different methodologies for constructing a program.

I'll refer to the first one as imperative, also procedural. That's kind of the most basic way you could think about writing a program. It's kind of a recipe. It's kind of like cooking with a recipe. Take this, add this, stir this, bake for 30 minutes, that kind of thing. So if you define a procedure, if you organize the way you think about it in terms of step by step instructions for, what should I do next? We refer to that kind of an approach as imperative.

We'll look at functional programming. There, even though you implement precisely the same algorithm, the perspective is a little bit different. There, rather than focusing quite so narrowly on the step by step, how do you get from A to B, the idea is going to be, think about the structure of the problem in terms of functions in a mathematical sense. By which I mean, functions that eat inputs, generate outputs, and don't have side effects. Side effects are things like setting variables that you can later look at.

Then I'll look at object-oriented programming. Again, you could be implementing precisely the same algorithm by using an object-oriented approach, but here the focus will be on building collections of data with procedures that are related, and organizing the solution to your problem in terms of a hierarchy of such structures.

So what I'd like to start off today with is to look at an example problem and how you

could program it using any of these three approaches. So the example program is going to be, find a sequence of operations-- by which I mean, an operation is either increment or square-- the idea is that the operations are things that we will do to integers. Find a sequence of operations, either increment or square, that transforms one integer, which is the initial value, i, into a goal, which I'll call, g.

So I want to think about the problem of finding such sequences. So for example, the sequence increment increment increment square, when applied to 1, would give 16. So I'm thinking about the first increment increments 1 to give you 2. The second increment increments 2 to give you 3. The third increment increments 3 to give you 4. Then square squares 4 to give you 16.

So I'll refer to this as having found the sequence of operations, increment increment increment square, that transforms 1 into 16. Everybody with me? OK, I'll be the judge of that.

So, to prove that you're with me, what's the minimum length sequence of increment and square operations needed to transform 1 into 100. You've got thirty seconds to discuss it with your neighbor, come to an agreement, and I'm gonna ask you to raise your hands with a number of fingers, (1), (2), (3), (4), or (5), indicating the right answer.

[AUDIENCE DISCUSSION]

PROFESSOR: OK, everybody raise your hand. Tell me a number of fingers equal to the right answer, raise your hand, show me a number of fingers. OK, I'm seeing-- OK. So keep in mind the number of fingers is the thing before the colon. That avoids the awkward way of saying less than 4. OK? So I want the number before the colon. So what is the minimum length sequence? Raise your hand, indicate a number of fingers. OK, the answers are improving. Higher so I can see you. OK, it's about 90% correct, I think.

OK, most people said (3), which is another name for 5. OK. So how do you get the answer 5? What's the answer? Somebody explain that answer? Yeah.

AUDIENCE: Increment increment square increment square?

PROFESSOR: That's exactly right. So since I have two operators, increments and squares, and since I'm trying to cover a big distance, 1 to 100, square increases faster-- at least for bigger numbers, than increment does. So what you'd like to do is figure out a way of coercing as many squares as possible. So a good thing to do is to start at the end, and you can take the square root of the first one evenly, and that gives you 10. But then you can't take the square root of 10, so you back off and you get 9, and then you can take the square root of 9, et cetera.

So there's any number of ways you could solve this problem, the point is, that there's a simple solution which is, the answer (3), which is a pseudonym for 5. So 5 operations will get you from 1 to 100.

So what I want to do now-- now that you know the answer to the question, I want to write a program to find that answer. The most straightforward approach that you could use is what we would call, imperative or procedural. The idea is to solve the problem by walking your way through it in some premeditated fashion.

A reasonable premeditated fashion would be, think about the sequences and order them by length. Think about all the sequences of length one, see if they work. Think about all the sequences of length two, see if they work. Three, see if they work.

And just keep going until you find one that works. That's a very reasonable-- that's a procedure. Start by thinking about short sequences, and proceed by making them longer and longer until you run into one that happens to solve your problem.

So that's what's going on here. First I gave a name to the operator, increment. Then I give a name to the operator, square-- that's just for convenience. Then what I want to do is write a program-- find sequence, that will start at some initial value, say 1, and go to some goal, say 100.

And the way I'll do that is to enumerate over the lengths of sequences that are possible -- 1, 2, 3, et cetera. I'll represent each one of those sequences of operations by this kind of a representation, I'll make a tuple that has a string that

tells me in English, what did I do? And an integer that tells me what the answer is after I've done that.

So looking ahead, this is the idea-- this is the program that I'm trying to construct. I would like the output of the program to start by thinking about all sequences of length one, then sequences of length two, then sequences of length three, then sequences of length four.

For each sequence of length one, I'd like to think about, what are all the possible sequences? Well, I could start with 1 and increment it to get 2. Or I could start with 1 and I could square it to 1. That's all the possible sequences of length one. Then I go on to length two, three, four, by the time I'm down here to length four, I could start with 1, increment square square increment, and that would give me 17. That's the structure of the program that I'm trying to make. Everybody with it?

So I'm going to define find sequence, I'm going to loop over all those lengths, I'm going to keep track of all the different sequences I found as a list of tuples. After last week you're supposed to be very comfortable with those kinds of words. Each one of the sequences is a tuple, a string and a final value, and the list is all possibilities.

And I'm going to try the ones of length one, then I'm going to append to each one to make sequences of two, and then I gonna append to that to make sequences of three, four, five, and keep on going. So the point is, that this is a very simple-minded, easy to conceive recipe for how to search through a large number of sequences and find the one with the minimum length. So when you write that program, it iterates down until it finds number 5, the 5-length sequence here it came up with the answer 100, so that's the answer.

The point is, that it was an easy way to write the program. We just think about telling somebody with a pencil and paper, what would you do? And we tell Python, rather than the person with the piece of paper, what to do. The approach is straightforward.

The only ugliness is that it ended up with three levels of loops. And the most

common kind of error in this kind of programming is you just botch the indices. Because it's got three nested layers of loops, it's very easy to lose track -- when you're thinking of layer three -- about what's happening in layer two.

So that's the only difficulty in this approach. The challenge is just to keep all the indices consistent. But it works. There's nothing wrong with this approach. It doesn't necessarily lead to the most modular designs, but you can write functional programs that work this way.

A different way to structure the program, I'll refer it-- here's a different version of the same program, the same algorithm, but I'm going to refer to this one as functional program. Here the idea is to focus on procedures that implement functions of a mathematical type.

I want to recast the problem, this time thinking about, how would I divide it up into functions that calculate things. So rather than focusing on, what's the set of nested loops that I have to make, I want to ask, what would be a meaningful calculation that I would want to do to solve this problem?

So the first thing I might do, focusing on this part, I might write a function, apply, where apply -- I'm using the methodology of functional programming -- apply is a pure function. It's going to eat some inputs, generate an output, and have no side effects.

So what I want it to do, I want to feed it a list of functions and ask it, what's the answer? So apply is going to have as its first argument a list, but the list is going to be a list of operations. It's not a list of strings, it's not a list of integers, it's a list of functions.

And what the procedure apply is going to do is step through that list, applying the functions one by one to argument. So the final goal is written down here. If I apply nothing to 7, the answer ought to be 7. If I apply increment to 7, the answer ought to be 8. If I apply square to 7, the answer ought to be 49. And if I apply increment square to 7, the answer ought to be 64.

There's a couple things you're supposed to see here. First off, apply is a pure function, it has no side effects. It eats its input, it does a calculation, it tells you an answer, and it's done. There's no extra things that's going on behind your back, it's not setting a variable, or creating a list, or doing anything like that. It has inputs, from those inputs it generates an output, and that's the end of the story.

Another thing that you should see is that I'm treating procedures as first-class objects. That's another tenet of functional programming. Rather than sending the input to apply as a list of text strings, or as a list of integers, I'm giving it a list of function names. I'm treating functions just as though it were any other kind of data. I'm making a list of functions just the same as I could make a list of integers, or if I made a list of strings, I'm just making a list of functions.

So another important feature of this program is the idea that the functions are being treated as first-class objects. The next procedure is addLevel. addLevel is going to be a second pure function. The idea there is that I'm going to use addLevel to construct all the different sequences of operations that are possible.

Each sequence is going to be a list, so the entire-- all possible sequences will be a list of lists. So the idea in addLevel was going to be that you take the list of lists and you generate from that list of lists a new list of lists that has one more element.

So if the first list of lists had four elements in it, how many lists are in the second list of lists? Five. Four, five? The first list of lists represents some number of different sequence of operations. My second list of lists-- what I want to do is think about all the different ways that could be extended from a length-4 sequence to a length-5 sequence.

How many ways can you extend a length-4 sequence to a length-5 sequence? Eight. So four becomes eight. Why does four become eight?

AUDIENCE: You've got like two possibilities.

PROFESSOR: Because there's two possibilities for the way that you could extend it. I could extend the length-4 sequence by adding on an increment operator, or I could extend it by

adding on a square operator. So every time I addLevel, it's going to create a new list of lists with double as many elements in it as the old list of lists. Does that make sense?

Other than that, the program works very much the same-- I should illustrate that. So addLevel applied to the list of lists, increment. I'm only considering one sequence of length-1, so how many ways can I change that if I'm willing to add procedures increment or square? Well, I could end up with increment increment by taking that one and that one. Or I could end up with increment square by taking that one and that one.

So if I were to extend the sequence, increment, two possible ways, by either incrementing again or by squaring again, I end up with two possibilities represented by this list of two different lists. Is that clear?

The important point is that I'm treating functions as any other kind of data element that could be added to a list. And when you run that program, you get an answer that's very similar to the program that we looked at previously, the imperative program. Except that the answer now is a list of functions.

So rather than generating a text-string like the previous example did, this program generates a list as the answer-- the program generates a list, which is a list of functions. The first element in the list is increment, the second element in the list is the function increment, the third element is square increment square.

So it gave you the same answer, it just represented it differently. Here I'm representing the answer as a list of functions. That make sense?

So there's a number of advantages to this approach. One of them is that it kind of automatically coerces you to think about modules. By specifically focusing my attention on, how could I break up that problem by defining functions? I end up with a bunch of functions here, apply and addLevel.

And being functions, they're modules. They are things that are easy to manipulate

in other contexts, and in particular, it's very easy to debug them. Especially in an interpretive environment like Python.

It's very easy to figure out if your program works by figuring out if each module works. That's one of the important features of modular programming. When you have a module, it means that you can use that module in multiple different ways. It's easy to break apart the debugging problem so that rather than trying to debug the one, monolithic structure-- which was the procedural program that we wrote in the first part, here we can debug the individual components, which happen to be functions.

So I can ask, what happens when I apply the empty list to 7? Well, my answer better be 7. So that provides a way of checking. So one of the features of thinking about the algorithm as being broken up into a number of functions is the greater modularity that allows you, for example, easier debugging.

A much more important reason for thinking this way though, is the clarity of thought. And that can be derived from another feature of the definition of apply. The particular way I defined apply is what we call recursive. It's recursive in the sense that the definition of apply calls apply.

OK, that sounds like a bad idea, right? How do I define something in terms of itself? The idea is that each application, every time apply calls itself-- the idea is to structure the procedure so that the new incarnation is in some sense, simpler than the previous one. If you can guarantee that, it will reduce the complexity of the problem from something that you don't know how to solve, to something that you do know how to solve. And for that reason, it represents a powerful way to think about structuring programs.

So as an example of that-- as an example of structuring programs as recursions, think about raising a number to a non-negative integer power. So if I'm trying to raise b to the n, if n is 0, and if n is a non-negative integer-- well, if n is 0, b to the n is 1. And if n is a non-negative integer, then b to the n can be reduced to b times b to the (n minus 1). Rewriting that functionally, if I say that my function b to the n can

be represented by f of n. This statement is precisely equivalent to saying that f of n is 1, if n is 0, or b times f of (n minus 1) if n is bigger than 0.

So the idea then, is that I may not know how to raise 2 to the 10th, but I can use the rule to say 2 to the 10th, oh, that must be 2 times 2 to the 9th. Great, I don't know how to do 2 to the 9th, either.

But 2 to the 9th is 2 times 2 to the 8th. And eventually, I boil it down to a case that I do know the answer to, in particular, b to the 0 is 1. I would express that idea in Python this way, define exponent of b, n. If n is 0, return 1, otherwise return b times exponent of b, (n minus 1).

OK, so what that does then when you invoke it-- invoking exponent of 2, 6, in fact, invokes exponent an additional six times. If I ask Python to evaluate exponent of 2, 6 it will end up calling exponent of 2, 5.

But to evaluate that, it will call exponent of 2, 4-- 3, 2, 1, 0. Then finally, it gets to the base case. When it gets to the call exponent of 2, 0 it falls into this case which returns 1 always.

So now exponent of 2, 0 returns 1, but then that can pick up where call exponent of 2, 1 left off. When I did 2, 1 it fell into this case, where it was trying to take 2 times exponent of 2, 0 Now it knows the answer to 2, 0 is 1. So I can multiply by 2 and get the answer is 2, and it backs out then all of the other answers.

So that's an example of how I could use recursion to reduce a complicated case to a base case-- a simple case that I know the answer to. Here, the recruitment with the power of n causes a linear increase in the number of invocations, so we would call that a linear process.

So the idea then is that recursion is a good way to reduce a complicated problem to a simpler problem. Now that algorithm wasn't particularly fast. If I imagine doing a big number like 1024. Raising a number to the 1024 power, that could take awhile. How long will it take? 1024 is going to-- it will make 1024 calls to reduce it to the base case.

Here's a way I could speed it up. Here I'm trying to make something called fast exponentiation, where I'm going to take advantage of another rule for the way exponentiation works. Not only is it true that I can write b to the n as b times b to the n minus 1, but if n happens to be even, there's another rule that I can use. If b is even, then I can raise b to the n over 2, and square the answer.

OK, that's more knowledge about the way exponents work. The point is, that when I think about the problem recursively, when I think about the problem as a function, there's a natural way to take advantage of that additional information.

So what I can do then is implement in Python in a scheme that looks very similar to the one that I used for the simple exponentiation, but it has a new rule embedded in there. So I can say, if n is 0 -- that was my original base case, if n is 0 the answer is 1. If the modulus of n when divided by 2 is 1, if it's odd, use the old rule-- which says b times the exponent of b, n minus 1.

However, if it's neither 0 nor odd, then use this new rule. If I use that procedure compared to the previous program, exponent, how many invocations of fastExponent is generated by a call to fastExponent of 2, 10?

So talk to your neighbor, figure out an answer, raise your hand. Use the number before the dot, just to keep you alert.

[AUDIENCE DISCUSSION]

PROFESSOR: So how many invocations of fastExponent is generated when you call it with the arguments 2, 10? Everybody raise your hand, tell me a number of fingers. OK, it's about 90%, something like that. The most common answer is five, how did you get five? So do you think of reducing fastExponent of 2, 10? What happens the first call?

AUDIENCE: It goes to [UNINTELLIGIBLE] divided by 2 is 5.

PROFESSOR: So the first thing that happens, is that it realizes that the 10 is even, so the 10 is not

0, it's not odd, it goes into this case. So it tries to run it with 5. Then when it tries to run fastExponent with 5, what happens?

OK, so 5 is not 0, so it's not the base case. It is odd, so 5 gets reduced to 4. 4 gets reduced to 2. 2 gets reduced to-- et cetera. So the idea is that we get a faster convergence because, just like in the very first example we did, we can do two different kinds of operations-- either decrement, or in this case half, and half works faster when the numbers are bigger than decrement does, so it's the same idea as in that first program.

So the idea, then, is that this requires 5 where we would have expected in the previous exponent procedure, it would have required 10. And that difference just gets bigger and bigger as we make n bigger and bigger.

Much more importantly though, than the fact that we can make something fast, is the idea that this is expressive. The idea is that by structuring the program this way, it was perfectly obvious how to incorporate more knowledge about the way exponents work. I could have done it by using a procedural approach. Do this, then do this, then do this. But then I have very complicated indices. Check if it's-- check if it's squared, check if it's whatever, and it's inside the loop, right? So I complicate my looping structure, where here-- because I'm using the functional approach, it's very easy to understand that it just introduces another case. It's nothing complicated.

So it's much more important-- the reason we think about recursion isn't so much that it's fast, it's because it's expressive. It's a way to incorporate knowledge about an answer into a solution, and that's what we'd like to do. We want to have a way of making complicated things simply.

Here's a way to incorporate knowledge about a procedure in a very straightforward fashion. All we have is a second case. Rather than making the loops more complicated, all we have is a simple new case in this statement.

And that's especially important when you think about harder problems. Here is a much harder problem. So Tower of Hanoi, you probably played with this as a kid. I

want to transfer all the disks from Post A to Post B-- I have a typo on your handout, by the way. I typed C on the handout, and then I wrote the program the other way. That should have been a B.

Transfer a stack of disks from Post A to Post B by moving the disks one at a time, never placing a bigger disk on top of a smaller disk. OK, that's kind of a challenging problem. It's a very challenging problem if you try to structure it as loops. If you try to structure it as loops I would recommend that you do this gazillions of times until you get the pattern stuck in your head, because the pattern's not obvious.

But the pattern is obvious if you think about it as a recursion. I don't know how to move n from Post A to Post B. How about this algorithm, take the top n minus 1, move those to C-- get them out of the way. I don't know how you're going to do that, but just trust me, just do it.

So move n minus 1 of them off the top, put them over on C, that exposes the bottom one. Move the bottom one to B, which is where I'd like it to be. And then by this mysterious process of moving n minus 1, bring the n minus 1 back from C back on top of B.

OK, what did I just do? I just started with a problem that I don't know the answer to. How do I move n disks from this post to that post? And I broke it into a sequence of three operations, two of which I don't know how to do, but one of which I know. The one will fortuitously be the base case-- or at least similar to the base case.

So the idea is, if I want to transfer the n, do two versions of the n minus 1 problem and glue them together with this middle step. OK, well I don't know how to do the n minus 1 problem either, just recurse. How do I do the n minus 1 one? Well, I'll do the n minus 2 one. I don't know how to do that either. Well, then do the n minus 3 one. Keep going until I get it down to 1. Because when I get down to 1, I do know how to move 1. Does that make sense?

This is supposed to illustrate that the power of recursion is in thinking. Some algorithms are easy to describe recursively. That set, is the set when the hard

problem can be reduced to a problem of the same type that is in some sense simpler. Here simpler means taking the index n, and turning it into n minus 1.

If you write this simple little Python snippet and run it, you get this procedure. And if you-- so the procedure means, take the top disk off A, put it on B. Take the top disk off A, put it on C. Take the top disk off B, put it on C.

If you do that procedure, it will magically do exactly the right thing. If you try to see the pattern here, so that you were to write this in a procedural approach, the pattern is very peculiar, but the algorithm for doing it is very simple. So the idea-- one of the reasons that we like recursion, is this idea of expressability. It makes it easy to express what we're trying to do.

OK, so that's a quick view of the first two ways to do the first problem that I talked about. We started thinking about a sequence of operations, either increment or square, that transforms an integer i into g.

The first thing we looked at was a procedural approach where I just gave you a recipe-- do this, do this, do this. The issue there is that I ended up with nested loops three deep. The issue there is keeping track of the indices for all of the different loops. We just talked about a functional approach. That can have an advantage, especially whenever the functional approach has a very simple implementation.

The last approach I want to talk about very briefly is object-oriented approach. Here-- I'm doing the same problem again, find the sequence of operations that transforms i into g, 1 into 100. Here I'm doing the same algorithm again. I mean, I haven't really changed the algorithm in going from the procedural approach to the functional approach, to here, the object-oriented approach.

I have changed the representation of the program, and that's the point. Here the representation of the program is in terms of objects. Here what I'm going to do is think about organizing all the sequences of operations that are possible in a tree.

So the sequences that I could do, I could do no operation followed by increment increment increment. Or I could do nothing increment increment square. Or I could

do nothing square increment square. So I'm thinking about all the possible sequences, but this time, I'm not thinking about it is a text-string, I'm not thinking about it as a list of functions that I need to call. I'm thinking about it as this tree, this tree of objects.

And in the object-oriented approach, what I'm going to do is think about the solution in terms of building up the tree until I find the answer. So the key in the object-oriented approach is to think of a useful object. What would I like to remember when I'm constructing the solution as this kind of a tree?

Well, if I call every circle a node, I can define a class that will keep track of all the important things about each node. And then I can structure my program as creating nodes, connecting them up into that tree, and looking for the answer. So each one of these nodes, each one of these circles, ought to have a parent, they ought to have an action, and they ought to have an answer.

So, for example, let's say that I started with the initial value i equals 1. Then this node right here might say, whatever your answer was at this point, the answer here was 1. So the answer here is going to be 1 after it's been incremented, so answer should be 2.

So associated with each one of these nodes is going to be a parent. The parent of this guy is this guy, the parent of this guy is this guy, the parent of this guy is that guy. Associated with each node is going to be an action. When I'm here, my action is to increment. When I'm here, my action is to square. And associated with each node is going to be the answer. So if I started at 1, by the time I get here, the answer's going to be 2, 4, 16.

So the idea is that I'm structuring the problem as a tree. Each node has a couple things it wants to remember, it wants to know who it's parent was. It wants to know, what's the operation associated with this node? And it wants to know the answer at this node. Also associated with each node, a method. And the method will be to print out the answer if this node is the answer.

So the print routine is going to have to say, OK, I ended up here and that happens to be the answer. Well, how did I get here? Well, I have a parent, who has a parent, who has a parent, who is none. So this routine is supposed to figure out the sequence of operations that led me from the initial value up at the top to the final answer that just happens to be goal. Is that clear?

So then I structure the program in terms of building those node structures. I start off by making a node, which is the top node. It has no parent, it has no action, and the answer to the top node is 1.

And then I just think through loops that are very similar to the loops that I was using before. They're structured similarly to the way they were structured in the case of the imperative program, except now the iterations are generating new nodes. The iterations are simpler, because each one just generates a node.

It's not like in the imperative case where I treated separately, what was the ASCII string that represented the answer, and what was the cumulative answer? Here I don't have two pieces of things that I need to do inside my loop, I just make a node.

So the idea is that when I solve the problem using this approach, what I'm trying to do is build a representation for the solution out of pieces that make sense for this problem.

So the important idea that I want to get across was to think about modularity at the next higher level. We started last time with modularity at the most microscopic level in Python, and we saw how Python had primitives that allowed us to build more complicated structures and simplified our task of understanding complex designs. We looked at the most primitive level in the first lecture.

What I just went over was how even within the structure of Python, you can structure programs that are more or less modular. And the important point is that the way you structure the program influences the degree of modularity that you get. So the structure of the program has a significant effect on how modular your solution is.

What I want to do for the rest of the time today is jump one higher level yet. So last time, primitives within Python. First half of this hour, imperative programming, functional programming, object-oriented programming, approaches that affect the modularity of your solution. Now I want to go up one more level and think about abstraction and modularity at a much higher level. And there, I'm going to think about programs that control the evolution of processes.

What's a process? A process is some set of procedures that evolve over time. An example was a bank account. OK, with the problems that we've been thinking about so far, there was a clear answer. What's the minimum-length sequence that blah, blah, blah. Right? There was a clear answer to that problem.

Here there isn't a clear answer to the problem, what's the answer to the bank account? The bank represents a process, it's something that changes with time. There are transactions. Every time you make a deposit or a withdrawal, you do a transaction that causes some action to happen in the bank. So there isn't an answer, there's an evolving answer. So those are the kinds of problems I want to think about now.

Graphical user interfaces. Again, there's no answer to Excel. Right, you can't say calculate the answer to Excel. Because whatever-- so the right answer for Excel depends on, what's the sequence of buttons that you're pressing, and what's the sequence of pull-down menus that you're doing, and all that sort of stuff.

So graphical user interfaces are processes. They're things that evolve over time. So they take a sequence of inputs, and they generate a sequence of outputs, just like a bank does.

Controllers. How do you make a robotic steerer? That's a process. There are inputs-- so for example, where are you in the middle of the lane? What time is it? How many pedestrians are you about to hit? There's all of those kinds of inputs.

And then there is things that you output-- which is like, keep going. So some of the steering rules are particularly simple. Other steering rules are you know, avoid the

guy this-- well, anyway.

So the idea is that when you were controlling processes, we're going to want to think about other kinds of programming structures. And even though the programming structures are going to be different, we're still going to want to have this idea of modularity. We're going to want to be able to think about, what are the primitives that we have, how can we combine them, what are the important abstractions, and how do we capture patterns? We're still going to want to do PCAP. But we're going to want to do PCAP at a much higher level.

And the programming module that I want to introduce is the idea of a state machine. A state machine is a very convenient structure to capture the behavior of a process. A state machine has an input, a state, and an output.

The idea is that it's a process. It's operated upon in steps. Every step there's a new input. From that input, the state machine can calculate an output and the next state. By keeping track of the state, this state machine can remember everything it needs to know so that it can continue the process into the future.

So the idea is that this is going to represent a fundamental programming element, that we'll be able to use to construct things that are like processes. And in particular, we use this to represent the calculations that are done within our robot. We'll think about the robot as being a process. It's not got an answer, you can't ask the question, what's the answer, to the robot.

But you can answer the question, if I were trying to steer and the sonar said (x,y,z), how fast should you turn the wheels? You can think about that as a state machine. There's some state of the robot. There are some inputs-- which is what the sonars are currently telling us, and there's some output-- which is how fast we're driving the wheels. So we want to think about this kind of a representation for processes that evolve over time.

OK, here's a very simple example, a turnstile. So we can think about the turnstile as a state machine. The turnstile has two states-- think about getting onto the T--

either it's locked, or it's not locked. Those are the two states of the turnstile.

There are some inputs. The inputs are, I put a coin in it, or I walk through it and the turnstile turns, or I don't do anything. And based on those inputs, and based on the current state, the turnstile has some outputs. So the turnstile can either allow you to enter, or ask you to pay. So the output might be an LED sign that says, please enter. Or it could have an LED sign that says, please pay. So those are out all of the elements of a state machine.

And so we can think about a graphical representation for the turnstile in terms of states. Here I'm representing the states as circles. There are two states in the turnstile, the turnstiles is either locked or unlocked. You move between states by getting inputs. So the inputs are represented by the arcs-- it's the top identifier on each arc is the input, the bottom identifier is the output, and the special arrow says I start here.

So the turnstiles start out locked in the morning-- the turnstile is locked. If you drop a coin in it, you'll move into the unlocked state and it will output enter. It will tell you that it's OK to enter because you've put a coin in it now. Then as long as you stay there and do nothing, it'll continue to say enter-- none enter. Or if you keep dropping coins in it, it'll say hey, keep going, and it'll continue to eat your coins and continue to say enter-- being very nice.

So in those cases it remains unlocked, it continues to tell you that you can enter, and that state persists until you turn the turnstile. When you turn the turnstile, this particular turnstile forgets how many coins you gave it and it simply locks and it tells you you've got to pay more.

So the idea is that we can capture the desired behavior of the turnstile-- the turnstile is a process, it's a thing that evolves over time. But we can capture the essence of what we would like the turnstile to do in terms of this diagram. We'll call this a state diagram.

And then we can think about behaviors in terms of a set of outputs as a function of

time. So imagine that as a function of time-- say I started in the state locked, and the input is nothing. So you wake up in the morning, you go to the T, overnight the turnstiles have all been locked. They all say that they're locked, they all say, pay, as their output.

If I don't do anything it just sits there and it says pay. If I drop a coin, it says enter and it moves into the unlocked state. If I don't do anything enter persists and it stays unlocked. If I walk through it, it eats the coin and it says, OK, you gotta pay now, and it locks.

So you can think about the evolution of this process-- the evolution of the turnstile in terms of the diagram with regard to time. And the idea then is that this is a kind of a representation that allows us to succinctly capture the important information about how processes work.

One of the important features of this representation is that it removes you one level away from the looping structure. Somewhere somebody's keeping track of the loop. Something's going from time 0 to time 1, to time 2, to 3 -- we'll talk about that later.

But for the time being, this representation focuses on what do you do at each instance in time? So it's a way of tearing apart the time part from the state transition part. So separate out the loop from the essence, it's very much like the functional example that I did.

The functions divorce themselves from the looping structure that we saw in the imperative approach, and you could define the functions independent of the looping. We're doing the same thing here. We can define the state machine representation independent of the looping.

And most importantly, this idea of state machines is modular. Let me show you how that plays out when we think about the robot. Think about the problem that I showed you last time where we're trying to have the robot go from point A to point B. Where, for example, it knows where it is -- say it has GPS or something. It knows-- from a map say, where it wants to be, but it has no idea of the obstacles between where it

is and where it wants to be.

So we can think about that kind of a problem-- we looked at this last time. It takes a step, it uses its sonars-- which are depicted by these lines, to infer that there are some walls-- that's the black dots. So it gets a reflection off the sonar that tells it that there's some obstacle in the way. That means it's not a good idea to try to plow straightforward, so if I back up, the original plan, it had no idea what was between it and its destination, so its plan was just buzz straight through.

At the end of its first step, it has got back sonar reports that tell it that there's a wall here, and here, and here, and here, and here, and here. And based on where the wall is, and the size of the robot, it can't get into any of these red squares. It's too big. So it knows its original path, which went from here to here, isn't going to work because it can't fit through. So it computes a new path, and then repeat.

So the idea is that we're solving a very complicated problem-- the robot 's solving a very complicated problem, and what I'd like to do now is think through a solution to that problem. As the robot's moving along, on every step it's getting some new input-- it's behaving just like a state machine.

On every step it gets new input from the sonars. Based on that new input, it knows where there's new obstacles. Based on that knowledge, it has a better idea of what the map is. Based on that knowledge, it has a better idea of finding a plan that might work. And then based on that knowledge, it knows how to tell the wheels to move forward.

So think about where it is right now. It has done, logically, three different things. It has figured out where the walls are. That's the thing represented in black and red. We'll call that -- it made a map. Given the map, it made a plan-- that's the blue thing. Given the plan, it figured out how to turn the wheels. It's going straight ahead.

So there are logically three things going on, all at the same time. You can imagine writing an imperative program to capture that behavior, but that's complicated. Not impossible, you would structure it as a bunch of loops. You know, it would-- unless

you're a crack programmer, and I'm sure you are-- it will end up being an ugly program.

The idea is, that by using the state machine representation we'll be able to think about how to pull apart those three pieces. So in particular, we can think about the state machine for the robot-- which would take the sensory input from the sonars and turn it into an action, which is turning the wheels-- we can think about that as having three modules.

There's the mapmaker module, which just looks at the sensory input. Doesn't care what the plan is, doesn't care how it's turning the wheels right now. All it the cares about is, what was the -- what's my current state? And what new information can I infer about the map from the sonars? So given the sonars and my current state, what's the most recent version of the map likely to look like.

Then there's a planner that says, given the sonars and the map, how would I construct this blue line? Given my current understanding of the map of the world, how would I get from where I am, to where I want to be? That's a plan.

Then from that plan-- say the robot is here, from the plan you can see that the thing the robot should do is move straight ahead. So the mover can take the heading-- the heading is just simply the coordinates of the first destination point in the plan. So the mover can look at the heading, the first destination point, and figure out how to make the wheels turn.

So the idea is that the state machine representation will let us parse a very complicated problem. A process-- the process by which the robot gets from here to there, a very complicated process. We're going to be able to break it up into pieces. The pieces are going to be much easier. They're going to be modular.

We will be able to write this piece, and debug it and make sure that it all works, independent of these other pieces. Because we just have it do any arbitrary walk around task or no walk around at all and ask whether the set of sensory inputs generates the correct map.

Similarly, this will be a module. This will be something that we can debug as well without having the other pieces working. We'll be able to say, OK, well, what if I fed a map that I hand constructed to the robot-- ignore this, just forget this, assume there is none of these-- I can hand make a map, feed it to this guy, give it sensory input and see if it comes up with the right heading.

It's just like in that functional approach example that I did. Having made the individual functions, we were able to test them individually because they were modules. Having made these individual state machines, we'll be able to test them individually because they're modules.

Then at the end we'll be able to paste them all together and get a very complicated behavior-- much more complicated than you would expect from the sum of the three programming activities. That's what we want. That's how we want to structure things. We want to be able to combine simple things and get something that seems more powerful than the linear combination, because it is.

Okay, so that's the goal. To do that, we're going to invent some new Python structures. And we're going to think about the representation of the state machine in Python and we will use objects. In fact, we use objects at three levels. We'll think about the highest level of abstraction of this problem as a state machine. So there's going to be state machine class.

The state machine class is going to have all the things in it that all state machines have to know about. All state machines, as we define them, are going to have to know how to start, how to make one step, and how to combine single steps into a sequence of steps-- and we'll call that operation of transduction.

So if you give me a list of inputs, transduce will make a list of outputs. So all state machines are going to be able to-- all state machines are going to have to know how to start, step, and transduce.

Then for particular types of state machines-- like the turnstile, we will generate a subclass. The subclass of turnstiles are going to have to know-- all of those are

going to have to do some other things. All turnstiles are going to have to have a rule for what happens when you're in state A and you get input i-- what should you do? Well there's going to be a general pattern that all turnstiles will subscribe to. We'll put those in the subclass, turnstile.

Then individual turnstiles. The first one in the Kendall stop, the second one in the Kendall stop, the first one in the Central stop, the third one in the Harvard stop-- particular turnstiles will be instances of the turnstile class.

So the way this plays out, we'll have a generic state machine class defined here. The state machine class will have a method, start. Start will create the instance variable, state. Every instance of a turnstile has to remember where it is, that's an instance variable. That instance variable gets created by the start routine.

So the very first thing you do if you want to talk about a new turnstile, is you have to instantiate the turnstile. Then you have to make an instance-- and you do that by calling start. Every state machine has to have a start routine. Every start routine will do this-- it'll create an instance variable.

Every state machine has to know how to take a step. We will make a generic method called, getNextValues, which will include all of the rules for how the class turnstile generates a new output, and a new state from the implant. And every subclass of state machine will have to define what one of those looks like.

The important thing here is that every state machine has to have a step routine, and the way we've decided to implement the step routine is to call the getNextValues, that is particular to the subclass of interest. So for example, the getNextValues for a turnstile, might be different from the getNextValues for a planner or a for a mapper.

Then we will subclass state machine according to the kind of state machine we're making. A planner, a mapper, a mover, a turnstile. So a turnstile will be a class, which is a subclass of state machine, and that's where we'll define how do you do getNextValues.

If you're a turnstile, here's the way you ought to do getNextValues. When I call

getNextValues, I'll tell you your current state and the current input, and from those two you ought to be able to figure out the next state and the output. So the getNextValues method for the turnstile class ought to take state and input-- the input, and the current state, and it generates a tuple, which is the new state and the output. And this is the rule by which it happens.

All turnstiles also have the same start state. So if you look back again at state machine, state machine created the instance variable, state, by looking at start's state. All turnstiles have the same start state, they all start locked.

And then finally, the way we'll use this, is we will instantiate a turnstile, we will make an instance of turnstile, put it into TS, turnstile, so this could be Central Square 16, for example. And then that instance, we'll be able to do things like start, step, and transduce. So if I transduced this the input, none, coin, none, turn, turn, coin, coin, on a particular turnstile, I'll generate some sequence of outputs like so.

So that's a complicated example using turnstiles. Let me do something simpler just to make sure you get the idea. Let's think about a simpler class, a subclass of state machines, a subclass called accumulator. That subclass always has a start state of 0, and has a getNextValues that returns the same value for the next state and for the output.

It always adds the current state to the input-- hence the name accumulator. It accumulates the input by taking the current state, adding the currently input to generate a new state-- that's just like a bank account. The bank account would accumulate transactions.

So if the input to the accumulator is 7, then the state gets incremented by 7, and the output is the value of the new state. So that's what this says, the starting state is always 0, the getNextValues always returns for this new state-- state plus input, and for the current output, the same thing as the current state. Everybody's clear?

So the question is, what would happen if I did this sequence of operations? Let's say that I have the state machine class, I have the accumulator subclass. I make an

instance A, I do some stuff to A, I make an instance B. I do some stuff to B, I do some stuff to A, and I type this. What gets printed? Take 30 seconds, talk to your neighbor, figure out what the answer is.

[AUDIENCE DISCUSSION]

PROFESSOR: OK. What will get printed by the print statement? Answer (1), (2), (3), (4), or (5). Everybody raise your hand with an answer. Excellent, excellent, almost 100%. I don't think I see a single wrong answer.

So everybody seems to be saying (2). (2), OK, so what's going on? So A is an accumulator, we start it, we step it. B is an accumulator, we start it, we step it, we step. So what is going to be A, how did you get five?

Well, A started at 0 that's what the start method does, right? Stepping it stepped it to 7. B made a new one, we can ignore that because it's a separate instance. The idea is that the state is associated with the instance.

So when we perturb the state of B by doing B.step, we are touching the state of A-- because they're instance variables. So that means when we decrement it, we're decrementing the 7 to get 5. OK, seems reasonable.

So the answer is either (1), or (2), or none. Why is it the same (21, 21) (21, 21), when we called the A.getNextValues, and here we're calling the B. Why are they the same?

AUDIENCE: Because it sets the state [INAUDIBLE].

PROFESSOR: So getNextValues is a pure function. So it gets passed in the state, the first number is the state, the second number is the input. So it doesn't pay any attention to the instance variable state, it uses the thing they got passed in.

Furthermore, the getNextValues associated with A, is precisely the same as the getNextValues associated with B, because getNextValues is something that's in the accumulator subclass. So they're exactly the same because it's the same method.

So what should be in your head, is a picture that looks like this. When we said create a state machine, that was something we put in our library, and that makes this environment. When we said subclass of state machine to make accumulator, that made a new class, a new environment.

And then when we made A and B by saying that A equals accumulator, and B equals accumulator, that made instances. The instances were initially empty, but when I say A.start, that created the state variable in the A instance, and in the B instance, and associated values with them.

So with the picture that you're supposed to have in mind, is that there's an environment associated with state machine. There's a separate environment associated with accumulator, but just one of those. But there's two instances. So there's two sets of instance variables, one associated with A and B. So the answer was number (2).

Now the robot example was supposed to motivate this idea that what we're going to do with state machines is put them together modularly. So what we're going to do-- and you start to do that this week, and also continuing into next week, we'll figure out how to compose state machine M1 and M2 in a cascade. That means the output of the first one becomes the input to the second one. We'll put things together in parallel, we'll put things together with feedback.

So what we'll do, is we'll figure out operators. PCAP-- primitives, means of combination, abstraction, patterns. The primitives are going to be the state machines, the ways we combine them are going to be these kinds of structures. Cascade them, put them in parallel, that kind of stuff. And that's going to allow us to make complicated brains out of simple ones.

Here's a very simple example, what if I had a state machine A, which is an accumulator, so A is an instance of accumulator. B is an instance of accumulator, there's two separate instances. And C is a new state composed by cascading A and B. What would happen if I typed out C.transduce so ([7, 3, 4])? What's the answer to C.transduce ([7,3,4])?

[AUDIENCE DISCUSSION]

PROFESSOR: So what happens when I transduce C? Well C is a composition of A and B. In particular, it's this composition. C is just the cascade of A into B. What that means is that if I put some sequence of inputs into the first machine, and if I put some sequence of the inputs into C, it's the same as putting it into A.

The output of the composite machine is the output of B. And the input to B is the same as the output of A. So the ([7,3,4]) goes through A, which is an accumulator, and comes out [7, 10, 14]. It accumulates, that's what accumulators do. That clear?

Then [7, 10, 14] goes into B and comes out [7, 17, 31]. So C, which is the cascade of two accumulators, ends up transforming [7, 3, 4] into [7, 17, 31].

So that's the idea for how we're going to compose complicated behaviors out of simple ones, and that's what the assignment is for this week.