PROFESSOR: OK, we're now, kind of on the home stretch, and we're entering the part of the course, that's actually my favorite part of the course. I can't promise it will be your favorite part of the course, but I hope so, at least for many of you. Throughout the term, we've been talking about ways to solve problems using computation. And one of the key lessons that I hope you're beginning to absorb is that we might use a completely different way to solve a problem with the computer than we would have used if we didn't have a computer handy. In particular, we might often use brute force, which you've never use with a pencil and paper, we might not do the mental gyrations required to try and formulate a closed form solution, but just guess a bunch of answers using successive approximation until we got there. A number of different techniques. And that's really what we're going to be doing for the rest of the term now. Except we won't be talking about algorithms per se, or not very much. Instead we'll be talking about more general techniques for using computers to solve problems that are actually hard. They don't just look hard, they in many cases really are hard. The plan of the next set of lectures is, I want to start with a simple example to give you a flavor of some of these issues. In the course of going through that example, I'll illustrate some both thinking tools and some software tools that can be used for tackling the example. Then abstract from the example to try and put in a more general framework. And then, dive back down and use that framework to tackle a series of other interesting problems.

Some things I would like you do think about learning along the way, is moving from an informal problem description to a more formal problem statement. So that's, in part, what we did with the optimization problems, right? We looked at an informal description about optimizing your way through the MIT curriculum, and then could formulate it using sigmas and other bizarre notation to try and formalize what we were really trying to do. And we did that as a preamble to writing any code. First understand the problem, formally, and then move on to the code. And we won't always be totally formal. I often like to use the word rigorous instead of formal. Implying that it won't look like math, per se, but it'll be precise and relatively unambiguous. So that will be one thing that I want you to think about as we go through these problems.

Another thing is inventing computational models. Almost every meaningful program we write is in some sense modeling the actual world. For writing a program to figure out how to keep a bridge from falling down, we're modeling the physics of bridges and wind and things like that. If we're writing a program to try and help us decide what stocks to buy or sell, we're trying to model the stock market in some sense. If we're trying to figure out who's going to win the Super Bowl, we're modeling football teams. But almost every problem, you build a piece of software, and you hope that it has some ability to mimic the actual situation you care about. And it's not the program we care about, per se, it's the world, and the program is merely a mechanism to help us understand the world better. And so we're going to ask the question, have we built a good model, in the computation, that gives us insight into the world?

As we do this, we'll see that we'll be dealing with and exploiting randomness. Depending upon your outlook of life, one of the sad things in the world or one of the happy things in the world, is that it's unpredictable. And things happen either at random or seemingly at random. It may be that if we had a deep enough understanding at the level of single atoms of the way the world works, we could model the weather and discover that, in fact, the weather patterns are not random but entirely predictable. Since we can't do that, we have to assume that they really are random, and we build models of the weather that assume a certain amount of randomness in it. Maybe if we understood the stock market well enough, we could have predicted the collapse in October. But since nobody does understand it well enough, when people model the stock market they assume that there's a certain amount of randomness, certain amount of stochastic things.

So, as far as we can observe the world, almost every interesting part of the world has randomness in it. And so when we build models, we will have to build models that are, if we want to be formal, we'll talk about a stochastic, which is just a fancy way of saying they incorporate randomness. So we haven't yet done that this semester, but almost everything we do from here on in, we will deal with a certain amount of randomness. What else are we going to be looking at? We'll be looking at the notion of making sense of data. As we look at, again, modeling the world, what we discover, there's a lot of data out there. If you work at Walmart and you're trying to decide what to stock on the shelves, you're building a model of what the customers might buy under certain circumstances, and that model is going to take account the entire history of what customer's have bought in the past. And that's, if you're Walmart, a lot of data. And so given that you have a lot of data about what's happened in the past, how can we interpret that, for example, to get insight into the future? And we do a lot of that. And so we will, for example, look at how can we draw pictures that help us visualize what's going on with the data, rather than trying to read say, a million numbers and inferring something from them. This is a big part of what people do with computers, is try and figure out how to understand large amounts of data.

And then finally, as we go through this last third of the course, I want to spend time in evaluating the quality of answers. It's easy to write a program that spits out a number, or string, or anything else. What's hard is to convince yourself that what it's spitting out is actually telling you the truth. And so we're going to look at the question about, you've written a program to do something relatively complicated, you get an answer out. You wouldn't have written the program if you knew what the answer was in advance, right? This is not like a high school physics experiment where you know what the answer should be. Here you went to the trouble of writing the program because you didn't know what the answer was. Now the program gives you an answer, should you believe it, or shouldn't you believe it? There are people who say, well, that's what the computer said, it must be true. By now you all have enough experience with programs that lie to know that's not the case. And so we'll be talking about how do you go about looking at the results, and deciding whether to believe them or not.

This is a lot of stuff, and we'll be coming back to this. I'll be skipping around from topic this topic. It may seem a little bit random, but there's actually a method to my madness. Part of it is, I believe that repeated exposure to some of these things is a good way to learn it. And so I'll be revisiting the same topic in more and more depth as we go, rather than taking a topic and exhausting it. So if you think about searches, this is more of a breadth first search than a depth first search.

All right. Think about these things as we now go through the next set of lectures. So, on for the first example. Consider the following situation: a seriously drunken university student, and I emphasize university as opposed to institute here, is standing in the middle of a field. Every second, he takes a step in some direction or another, but, you know, just sort of pretty random, because he's really out of it. But the field is constrained that for the moment we'll assume that, well, we'll come back to that. Now I'm going to ask you just a question. So you've got this student who, every second or so, takes a step in some direction or another. If the student did this for 500 seconds or 1000 seconds, how far do you expect the student would be from where he started? And I say he because most of the drunks are, of course, males. Anybody want to, what do you think?

STUDENT: Back where he started.

PROFESSOR: So we have a thing that, on average, and of course since there's randomness it won't be the same every time. Pretty much where he started. Let me ask a question. So if you believe that, you believe he'd be probably the same distance in 1000 seconds as in 500 seconds. Anyone want to posit the counterposition? That in fact, the longer the clock runs, the further the student will be from where he started? Nobody? All right, what do you think?

STUDENT: [INAUDIBLE]

PROFESSOR: All right, that's a good answer. So the answer there was, well, if you asked for your best guess of where the student was, the best guess is where the student started. On the other hand, if you ask how far was the student from where he started, the best guess wouldn't be zero. That's a great answer, because it addresses the fact that you have to be very careful what question you're asking. And there are subtle differences between those two questions, and they might have very different answers. So that's part of that first step, going from an informal description, to trying to formalize it or be more rigorous about what is exactly the problem you're trying to solve.

All right, we'll take a vote. And everyone has to vote here, because it's not a democracy where you're allowed to not vote. And the question I'm asking is, is the expected difference, does the expected distance from the origin grow over time, or remain constant, roughly? Who thinks it grows over time? Who thinks it remains constant? Well, the constants have it. But just because that's the way most people vote, it doesn't make it true. Now let's find out what the actual truth is.

All right. So let's start by sketching a simple model of the situation. And, one of the things I want to stress as we do these things, in developing anything of this nature, start simple. So start with some simple approximation to the real problem. Check that out, and then, if it turns out not to be a good enough model of the world, add some complications, but don't start with the complicated model. Always start with the simple model. So I'm going to start with the simple model. And I'm going to assume, as we've seen before, that we have Cartesian coordinates and that the player is, or the drunk is, standing on a field that has been cut to resemble a piece of graph paper. They got the groundskeeper from Fenway Park or something, there, and it looks like a beautiful piece of paper. Furthermore, I'm going to assume for simplicity, that the student can only go in one of four directions: north, south, east, or west. OK, we could certainly generalize that, and we will, to something more complicated, but for now we'll keep it simple. And we'll try and go to the board and draw what might happen.

So the student starts here and takes a step in one direction or another. So as the mathematicians say, without loss of generality, let's just assume that the first step is here. Well what we know for sure, is after one step the student is further from the origin than at the start. But, that doesn't tell us a lot. What happens after the second step? Well, the student could come back to the origin and be closer, that's one possibility, the student could go up here, in which case the student is a little further, the student could go down here, in which case the student is a little further, or the student could go over here, in which case the student is twice as far. So we see for the second step, three times out of four you get further away. What happens in the third step? So let's look at this one. Well, the student could come here, which is closer, could come here, which is closer, could go there, which is further, or could go there, which is further. So the third step, with equal probability, the student is further or closer if the student is here. And we could continue. So as you can see, it gets pretty complicated, as you project how far out it could always get. All right, and this is symmetric to this, but this is yet a different case. So, this sort of says, OK, I'm going to get tired of drawing things on the board.

So, being who I am, I say, let's write a program to do this. And in fact, what I'm going to write a program to do, is simulate what is called a random walk. And these will be two themes we're going to spend a lot of time on. Simulation, where we try and build the model that pretends it's the real world and simulates what goes on, and a random walk. Now, I'm giving you the classic story about a random walk which you can visualize, at least I hope, but as we'll see, random walks are very general, and are used to address a lot of real problems. So we'll write this program, and I want to start by thinking about designing the structure of the solution. Because one of the things I'm trying to do in this next set of lectures is bring together a lot of the things we have talked about over the course of the semester, about how we go about designing and building programs. Trying to give you a case study, if you will.

So let's begin in line with what Professor Grimson talked about, about thinking about what might be the appropriate data abstractions. Well, so, I think it would be good to have a location, since after all the whole problem talks about where the drunk is. I decided I also wanted to introduce an abstraction called compass point to capture the notion that the student is going north, south, east, or west. And later, maybe I'll decide that needs to be more complicated and that the student can go north by northwest, or maybe all sorts of things. But that it would probably pay to separate the notion of direction, which is what this is, from location. Maybe I should have called this direction instead of compass point, but I didn't. But, I thought about the problem and said, well, these things really are separate locations. Where you are and where you might head from there are not the same, so let's separate them. Then I said, well of course, there is this notion that the person is in the field, so maybe I want to have field as a separate thing. So think of that as the whole Cartesian plain, as opposed to a point in the plane, which is what the location is. And finally, I better have a drunk, because after all this problem is all about drunks.

All right, so we're now going to look at some code. I made this code as simple as I could to illustrate the points I wanted to illustrate. This means I left out a lot of things that ought to be included in a good program. So I want to just warn you. So for example, I've already told you that when I build data abstractions, I always put in an underbar underbar str function so that I can print them when I'm debugging. Well, you won't see that in this code, because I felt it just cluttered the code up to make the points. You'll see less defensive programming than I would normally use. But again I wanted to sort of pare things down to the essence, since the essence is, all by itself, probably confusing enough.

All right. So let's look at it. You have this on your double-sided handout. This is on side 1one So at the top, you'll see that I'm importing three things. Well, math you've heard about. And I'm importing that because I'm going to need a square root. Random, you haven't heard about. This is a package that lets me choose things at random, and I'll show you some of the ways we can use it, but it actually provides something that technically is pseudo-random. Which means that as far as we can tell, it's behaving randomly, but since the computer is in itself a deterministic machine, it's not really random. But it's so close to random that we might as well pretend it is, you can't tell that it isn't. So we'll import random. That will let us make random guesses of things like, give me a random number between 0 and 1. And it will give you a random number. And finally something called Pylab. Anyone here use Matlab? All right, well then, you'll find Pylab kind of comforting. Pylab brings into Python a lot of the features of Matlab And if you haven't used Matlab, don't worry, because we'll explain everything. For the purposes of this set of lectures, the next few lectures, at least, the only thing we're going to get out of this is a bunch of tools for drawing pretty graphs. And we won't get to those today, probably, so don't worry about it. But it's a nice package, which you'll be glad to learn.

OK, now let's move on to the code, which has got things that should look familiar to you. And I know there used to be a laser -- ah, here's the laser pointer. So we'll see at the top location, it is a class. By now you've gotten as familiar as you want to be in many senses, with classes. It's got an underbar underbar init that gives me, essentially, a point with an x-coordinate and a y-coordinate. It's got get coords, the third function, or method, and what you can see what that does is it's returning a tuple of two values. I could have had it get x and get y, but it turned out, in fact, my first iteration of this it did have a get x and get y, and when I looked at the code that was using it, I realized whenever I got one, I wanted both, and it just seemed kind of silly, so I did something that made the using code a little bit better. It's got get distance, the last method. You saw this in Professor Grimson's lecture, where he used the Pythagorean theorem to basically compute the distance on a hypotenuse, to tell you how far a point was from the origin. That's why I wanted math. And then the one thing it has that it's unlike the example you saw from Professor Grimson, was up here, this move. And this basically takes a point, a location, and an x- and a y- coordinate, and returns another location, in which I've incremented the x and the y, perhaps incrementing by 0. And so now you can see how I'm going to be able to mimic 1 step or any number of steps by using this move. Any questions about this?

Great, all right. The next one you'll see is compass point. Remember, this was the abstraction that was going to let me conveniently deal with directions. The first thing you'll see is, there's a global variable in the sense that it's external to any of the methods, and you'll note it's not self dot possibles, but just possibles. Because I don't want a new copy of this for every instance. And this tells me the possible directions, which I've abbreviated as n, s, e, and w. And I leave it to you to figure out what those abbreviations stand for. Then I've got init, which takes a point and it first checks to see if the point is in self dot possibles. Should I have bothered saying self? Do I need to write self there? This is a test of classes. It'll work. What do you think? Who thinks I need to write self, and who thinks I don't? Who thinks I need to write self, raise your hand? Who thinks I don't need to write self? The don't needs have it. So in an act of intense bravery, since I have not run it without this, we'll see what happens when it comes time to run the program.

So if point is in possibles, it will take self dot p t will be assigned p t. Otherwise, I'll raise a value error. And this is a little piece of programming, and what I typically do at a minimum, when I raise these exceptions, is make sure that it says where it's coming from. So this says it's gonna raise this value error in the method compass point dot underbar underbar init, underbar underbar. This is so when I run the program, and I see a message, I don't have to scratch my head figuring out where did things go wrong? So just a convention I follow a lot. And then I've got move here. And I'll move self some distance, and you can see what I'm doing. If self is north, I'm going to return 0 in distance. So I'm now getting a tuple, which will basically be used to say, all right, we're going to implement x by 0 and y by distance, which is what you think of about moving due north. And if it's south, we'll increment x by 0, and increment y by minus distance, heading down the graph. And similarly for east and west. And in the sad event I call this with something that's not north, south, east, or west, again I'll raise an exception. OK, little bit at a time. So I hope that nothing here looks complicated. In fact, it should all look kind of boring.

Now we'll get to field, which should look a little less boring. So now I've got a field. And, well, before I get to field, I want to go back to something about the first 2 abstractions: compass point and location. Whenever we design one of these abstractions, we're making some things possible and some things impossible. We're making decisions, and so what decisions are encapsulated in these first two classes? Well, one decision is that it's a 2-dimensional space. I basically said we've got x and y and that's all we've got. This student cannot fly. The student cannot dig a hole and go into the ground, we're only moving in 2 dimensions, so that's fixed. The other decision I made is that this student can only move in 1 of 4 directions. So, couple of good things here. One, I've simplified the world, which will make it easier for us to see what's going on. But two, I know where I've made those decisions, and later if I want to go back and say, you know it'd be fun if the student could fly, let's say we're a drunken pigeon instead of a drunken student. Or it would be a good idea to realize that, in fact, they might head off at any weird angle. I know what code I have to change. It's encapsulated in a single place. So that's why possibles is part of compass point, rather than scattered throughout the program. So this is nice way to think about it.

All right, let's move on and look at field. So field, it's got an init, it takes a self, a drunk, and a location, and puts the drunk in the field at that location. It can move. So let's look at what happens when we try and move self in some direction, in some distance. We say the old location is the current location, and then x c and y c, think of that as x-change and y-change, get c p dot move of dist. Where is it gonna find c p dot move? It's going to use compass point dot move, because as we'll see, c p is an object of type, of class, compass point. So this is a normal and typical way we structure things with classes. The move of one class is defined using the move of another class. This gives us the modularity we so prize. And then I'll say self dot location is old loc dot move of x c and x c. Now this is oldloc dot move is going to get the move from class location. And you'll remember, that was the thing that just added appropriate values to x and y. So I'll use compass point to get those values, and then oldloc to get the new location. Then I'll be able to get the loc and I'll be able to get the drunk.

So, let me ask the same question about field I've asked about the others? What interesting decisions, if any, have I embodied in the way I've designed and implemented this abstraction? There's at least one pretty interesting decision here. Just interesting to me because my first version was far more complicated. Well, how many drunks can I have in a field at a time? We have an answer which will not be recorded, because it was merely raising a finger, but it happened to be the correct number of fingers: one. And it was this finger that got pointed at me. We've embodied the fact that you can have one drunk, exactly one drunk in the field. This is a solitary alcoholic. Later we might say, well, it would be fun to put a whole bunch of drunken students in and watch what happens when they bump into each other. And we'll actually later give you a problem set, not with students, but with other things where there might be more than one in a field. But here, I've made that decision. But again, I know where I've made it, and if later I go back and say let's put a bunch of them in, I don't have to change compass point, I don't have to change location, I only have to change field. And we'll also see, I don't have to change drunk. So again, it's very nice that the class structure, the modularity let's us have decisions in exactly one place in our code. Usually important.

All right, now let's look at drunk. So the drunk has a name. And, like everything else, a move operation. This is the most complicated and interesting move. Because here is where I'm encapsulating the decisions about what the drunk actually does. That the drunk, for example, doesn't head north and just keep on going. So, let's look at what happens here. It's got three parameters: self, the field the drunk is moving in, and time. How long the drunk is going to move. And you may notice something that you haven't seen before, at least some of you, time equals one is sitting up there. There. Python allows us to have what are called default values for parameters. What this says is that if I call this method without that last argument, rather than getting an error message saying it expects three arguments it only got two, it chooses the default value, in this case one for the third argument. This is actually a pretty useful programming paradigm, because there's often a very sensible default. And it can simplify things. It's not an intrinsically interesting or important part of what I'm showing you here. The reason I'm showing it is, you'll be getting a problem set in which default values are there because we're using some other things. And as you bring in libraries and modules from elsewhere, you'll find that there are a lot of these things. And in fact a lot of the functions you've already been using for the built-in types happen to have default values. For example, when you look at things in the init part of a range, it's actually choosing, say, 0 as a start. But don't worry about it. This just says if you don't pass it a time, use 1.

And then what it does, is it says, if field dot get drunk is not equal to self, raise an exception. OK, you've asked me to move a drunk, it doesn't happen to be in the field. That's not very interesting. But now we come to the interesting part. For i in range time, here by the way range does have a default value of 0, and since I didn't supply it, it used it. Point equals compass point, and here's the most interesting thing, random dot choice of compass point dot possibles. Random dot choice is a function in the random module which we've imported and it takes as an argument a sequence, a list of some sort, a container, and it picks a random element out of that. So here we're passing it in, four possible values, and it's just going to pick one of them at random. So you can see by my having encapsulated the possibles array in compass point, I don't have to worry in drunk about how many possible directions this person could head off in. And then it calls field dot move, with the resultant point, and in this case one. All right? We've now finished all of this sort of groundwork that we need to go and actually build the simulation that will model our problem. So let's look at how we use all of this. I've now got a function perform trial, which will take a time in a field. So I'll get a starting point of f dot getloc, wherever the drunk happens to be at this point in time in the field. And then for t in range 1 to time plus 1, I'm going to call field dot get drunk, that will give me the drunk that's in the field, and then I'll get the move function for that drunk. So what you see here is I can actually, in this dot notation, take advantage of the fact, where are we, we're down here, I've got to get far enough away that I can see it -- Where was I?

STUDENT: You were in the right place right there, down slightly.

PROFESSOR: -- down slightly, here, right. I've called the function, which has returned an object, f dot get drunk returns an object of class drunk. And then I can select the method associated with that object, the move method, and move the drunk. This gets back to a point that we've emphasized before, that these objects are first class citizens in Python. You can use functions and methods to generate them, and use them just as if you'd typed them. So that will get me that, and then I'll call the move of that drunk with the field. Then I'll get the location, the new location, and then I'll say, distance equals new loc dot get distance of start, how far is the new location from wherever the starting location was? I'll append it to a list, and then I'll return it. So now I have a list, and in the list I have how far away the drunk is after each time step. I'm just collecting the list of distances. So I don't have a list of locations, I don't know where, I can't plot the trajectory, but I can plot the distances.

OK, now, let's put it all together. So I'll say the drunk is equal to, is drunk of Homer Simpson. I had thought about using the name of someone on campus and decided that, since it was going to be taped for OpenCourseWare, I didn't want to go there. Sometimes I lack courage. Then for i in range 3, and all this says is I'm going to test, I'm going to run the simulation three different times, f is equal to field of drunk and location 0, 0, starting in the middle of the field. Distances equal perform trial 500, 500 steps on that field, and the rest of it is magic you don't need to understand because we'll come back to next lecture, which is, how do I put all this in a pretty picture? So ignore all of that for now, and that's just the Pylab stuff for plotting the pictures. All right, we're there, let's run it. Huh. Remember the change we made? Guess what? Well, we know how to fix that. And we'll come back to it and ask what's really going on here. What does self refer to here? It refers to the class compass point, rather than an instance of the class. Again, driving home the point that classes are objects just like everything else.

STUDENT: [INAUDIBLE]

PROFESSOR: Oh, sorry. Thank you. It did it have in here, right. All right, now let's run it, see what we get. We get a picture. Well, so we see that, at least for these three tests, the majority of the public was wrong. It seems that the longer we run it, the further from the origin Homer seems to be getting. Well, let's try it again. Maybe we just got unlucky three times. You can see from the divergence here, that, of course, a lot of things go on. I hope you can also see the advantage of being able to plot it rather than having to look at those arrays. Well, we get different answers, but you know what? The trend still seems pretty clear. It does seem over time, that we wander further away. I know, further away. So, looks at least for the moment, that perhaps we were wrong.

Well, this is rather silly for me to keep running it over and over again. So clearly what I ought to do is, do it in a more organized way. So now if we go back, and see the, kind of the right way to do it, I'm going to get rid of this stuff here, which was used just to stop the previous computation. I'm going to write a function called perform sim. And what that does, is it takes the time and the number of trials that we wanted to do. It takes distlist equals the empty list in this case, to start with, and then it is basically going to call perform trial, the function we already looked at over and over again. And get a whole bunch of lists back, a list each time it calls it. And then, compute an average. Well, not yet, so it says, it says d equals drunk, field start in zero, distances equals perform trial, append the new list, and what it does, is it returns a list of lists. Where each list is what we had for the previous trials. Make sense? And then I'm going to have 1 more function call, answer a question. Which takes the maximum time and the number of trials, and it's going to do some statistics on all of the lists. We'll come back to this a week from today. If you are, have absolutely nothing to do for the next week, I'll give you a hint. I have salted a bug here in this latest code, and you might have some fun looking at what that bug is.