JOHN GUTTAG: Today we're starting a new topic, which is, of course, related to previous topics. As usual, if you go to either the 60002 or the 600 web site, you'll find both today's PowerPoint and today's Python. You'll discover if you look at the Python file that there is quite a lot of code in there. And I'll be talking only about some of it. But it's probably all worth looking at. And a fair amount of reading associated with this week.

Why are we looking at random walks? See a picture here of, think of them as molecules just bouncing around. This is actually a picture of what's called Brownian motion, though Robert Brown probably did not discover it. We're looking at random walks because, well, first of all, they're important in many domains. There are people who will argue, for example, that the movement of prices in the stock market is best modeled as a random walk. There was a very popular book called *A Random Walk Down Wall Street* that made this argument. And a lot of modern portfolio analysis is based upon that.

Those of you who are not interested in making money, and I presume that's most of you, it's also very important in many physical processes. We use random walks, say, to model diffusion, heat diffusion, or the diffusion of molecules in suspension, et cetera. So they're very important in a lot of scientific, and indeed, social disciplines.

They're not the only important thing, so why are we looking at those? Because I think it provides a really good illustration of how we can use simulation to understand the world around us. And it does give me an excuse to cover some important topics related to programming. You'll remember that one of the subtexts of the course is while I'm covering a lot of what you might think of as abstract material, we're using it as an excuse to teach more about programming and software engineering. A little practice with classes and subclassing, and we're going to also look at producing plots.

So the first random walk I want to look at is actually not a diffusion process or the stock market, but an actual walk. So imagine that you've got a field which has somehow inexplicably been mown to look like a piece of graph paper, and you've got a drunk wandering around the field, taking a step every once in a while in some random direction. We can then ask the question is there an interesting relationship between the number of steps the drunk takes and how far the drunk is from the origin at the end of those steps? You could imagine that if the drunk takes more steps, he's ever further from the origin. Or maybe you could imagine, since it's random, that he just wanders away and he wanders back in all directions and more or less never gets very far.

So just out of curiosity, I'll take a poll. Who thinks that the drunk doesn't much matter how many steps he takes, he'll be more or less the same distance away? And who thinks the more steps he takes, the further away he's likely to be? It seems to be a season where when you take polls, they come out almost tied.

Let's look at a small example. Suppose he takes one step only. Well, if he takes one step, and we'll assume for simplicity that he's not so drunk that he moves at random. He either moves north or south, east or west. These are all the places he can get to in one step. What they have in common is that after one step, the drunk is always exactly one unit away from the origin.

Well, how about after two steps? So without loss of generality, let's assume that the first step-- let me use the pen that you're supposed to use to write on this, rather than this pen, which would make a real mess on my screen. What did I do with it? Well, I won't write on it.

So without loss of generality, we'll assume that the drunk is there after one step. Took one step to the east. Well, after two steps, those are all the possible places he could be. So on average, how far is the drunk from the origin? Well, if we look, he could either be two steps away, if he took another step east, zero steps away, if he took a step west, or what do we see for the top two? Well, the top and the bottom one, we can go back and use the Pythagorean theorem. c squared equals a squared plus b squared. And that will tell us that it'll be the square root of a squared plus b squared. And that will tell us how far away the upper two are, and then we can just average them and get a distance. And as we can see, on average, the drunk will be a little bit further away after two steps than after one step.

Well how about after 100,000 steps? It would be a little bit tedious to go through the case analysis I just did. There are a lot of cases after 100,000 steps. So we end up resorting to a simulation. So we'll structure it exactly the same way we've been structuring our other simulations. We're going to simulate one walk of k steps, n such walks, and then report the average distance from the origin of the n walks.

Before we do that, in line with the software engineering theme of the course, we'll start by defining some useful abstractions. There are three of them I want to look at. Location, the field that the drunk is in, and the drunk him or herself.

So let's first look at location. This is going to be an immutable type. So what we see here-- as long as I can't point in the screen, I'll point with a pointer-- is that we'll initiate it. We'll initialize it with an x and y value. That makes sense. We'll be able to have two getters, getX and getY. And here's how we see it's immutable. What move is doing is it's not changing the location, it's returning a new location. Perhaps move is poor choice of name for that. But that is what it's doing. It's just returning a new location where it adds the change in x and the change in y to get two new xy values.

Notice, by the way, that I'm not restricting these to be integers, or one, or anything like that. So this would work even if I did not want to take those nice little east-west, north-south steps. I've got a underbar underbar string, _str_, and then here's my implementation-- you can see it's very sophisticated-- of the Pythagorean theorem. So I just do it that way, and that will get me the distance between two things.

It's one of the annoying things about classes in, actually, all languages I know with classes, is you would like to think that self and other-- there's a symmetry here. The distance from self to other is the same as from other to self. But syntactically, because of the way the language is structured, we treat them a little bit differently.

How about class Drunk? Well, this is kind of boring. Drunk has a name and a string. And that's all. The point of this, and I don't think we've looked at this before, is this is not intended to be a useful class on its own. It's what we call a base class. The notion here is its only purpose is to be inherited. It's not supposed to be useful on itself, but it does give me something that will be used for the two subclasses.

And we'll look at two subclasses. The so-called usual drunk, the one I tried to simulate when I was wandering around, wanders around at random. And a drunk I like to think of it as a New Englander, or a masochistic drunk, who tries forever to move ever northward, because he or she wants to be frozen. I do like this picture of entering the state of Maine in the winter.

So here is the usual drunk. Subclass of drunk, and it can take steps at random, one step, either increasing y, a step north, decreasing y, a step south, increasing x, a step east, or decreasing x a step west. So those are the choices. And it's going to return one of those at random. I think we saw random.choice in the last lecture.

And then our masochistic drunk, it's almost the same, except the choices are slightly different. If he chooses to head north, he doesn't go one step. He goes 1.1 steps north. And if he chooses to go south, he only goes 9/10 of a step. So what we're seeing here is what's called a biased random walk. And the bias here is the direction of the walk that he's moving either up or down. Pretty simple.

How about just for to test things out, we'll ask the question is this an immutable or a mutable type? Are drunks mutable or immutable? This is a deep philosophical question. But if we ignore the philosophical underpinnings of that question, what about the two types here? Who thinks it's immutable? Who thinks it's mutable?

Why do you think it's mutable? What's getting changed? The answer is nothing. It gets created, and then it's returning the step, but it's not actually changing the drunk.

So so far we have two things that are immutable, drunks and locations. Let's look at fields. Fields are a little bit more complicated. So field will be a dictionary, and the dictionary is going to map a drunk to his or her location in the field. So we can add a drunk at some location, and we're going to check. And if the drunk is already there, we're not going to put the drunk in. We're going to raise a value error, "Duplicate drunk." Otherwise we're going to set the value of drunkenness mapping to loc.

Now you see, by the way, why I wanted drunks to be immutable. Because they have to be hashable so I can use them as a key in a dictionary. So it was not an idle question whether they were immutable. It was an important question.

I can get the location of a drunk. If the drunk is not in there, then I'll raise a different value error, "Drunk not in field." Otherwise I'll return the location associated with that drunk.

And finally, we're going to have moveDrunk. Again I'll check whether the drunk is there. If the drunk is there, I'm going to get the distance on x and the distance in y by calling drunk.takeStep. So we saw takeStep for a drunk didn't move the drunk anywhere, because the drunks were immutable, but returned new locations. A new x and new values. And then I'm going to use that to move the drunk in the field. So I'll set self.drunk, so drunk to move x distance and y distance.

So it's very simple, but having built this set of classes, we can now actually write the simulation. Oh. What about our classes? Are they mutable or immutable? Not classes. What about fields? Any votes for mutable? Yeah, exactly. Because you can see I'm mutating it right here. I'm changing the value of the dictionary. And in fact, every time I add a drunk to the field, I'm changing the value of the dictionary, which is to say mutating the field. So I'll have a bunch of locations, which are immutable objects. Makes sense that a location is immutable. A bunch of drunks, and the thing I'm going to change is where the drunks are in the field.

I said we'd start by simulating a single walk. So here it is, a walk in a field with a drunk, and that drunk will take some number of steps in the field. And you can see this. It's very simple. I just have a loop. Drunk takes some number of random steps, and I'm going to return the distance from the start to the final location of the drunk.

So how far is the drunk from the origin? I then need to simulate multiple walks. Notice here that I've got the number of steps, the number of trials, and dClass stands for class of the drunk. And that's because I want to use the same function to simulate as many different kinds of drunks as I care about. We've only seen two here, the masochistic drunk and the usual drunk, but you can imagine many other kinds as well.

So let's do it. So here I'm going to simulate a walk for one drunk, Homer. So we'll create a drunk named Homer, or the variable Homer, which is the drunk class. Then the origin, distances, and for t in range number of trials, we'll just do it, and then we'll return the distances. So it's initialized to the empty list. So we're going to return a list for however many trials we do, how far the drunk ended up from the origin. Then we can average that, and we look at the mean. Maybe we'll look at the min or the max. Lots of different questions we could ask about the behavior.

And now we can put it all together here. So drunkTest will take a set of different walk lengths, a list of different walk lengths, the number of trials, and the class. And for a number of steps and walk lengths, distances will be simWalks of number of steps, numTrials, dClass. And then I'm going to just print some statistics.

You may or may not have seen this. This is something that's built in to Python. I can ask for the name of a class. So dClass, remember, is a class, and _name_ will give me the name of the class. Might be usual, it might be drunk, in this case. So let's try it.

So the code we've looked at. So let's go down here, and we'll run it, and we'll try it for walks of 10, 100, 1,000, and 10,000 steps. And we'll do 100 trials.

Here's what we got. So my question to you is does this look plausible? What is it telling us here? Well, it's telling us here that the length of the walk actually doesn't really affect-- the number of steps doesn't affect how far the drunk gets. There's some randomness. 8.6, 8.57, 9.2, 8.7. Not much variance. So we've done this simulation and we've learned something, maybe.

So does this look plausible? We can look at it here. I've just transcribed it. What do you think? Well, go ahead.

AUDIENCE: I was going to say, it seems plausible because after the first two steps, there's a 50% chance he's going closer to the origin. And a 50% chance he's going away from it.

JOHN GUTTAG: So we have at least one vote for plausible, and it's certainly a plausible argument. Well, one of the things we need to learn to do is whenever we build a simulation, we need to do what I call a sanity check to see whether or not the simulation actually makes sense. So if we're going to do a sanity check, what might we do in this case? We should try it on cases where we think we know the answer.

So we say, let's take a really simple case where we're pretty sure we know what the answer is. Let's run our simulation and make sure it gives us the right answer for this simple case. So if we think of a sanity check here, maybe we should look at these numbers. We just did it. We know how far the drunk should get in zero steps. How far should the drunk move in zero steps? Zero. How far should the drunk move in one steps? We know that should be one. Two steps, well, we knew what that should be.

Well, if I run this sanity check, these are the numbers I get. I should be pretty suspicious. I should also be suspicious they're kind of the same numbers I got for 10,000 steps. What should I think about? I should think that maybe there's a bug in my code.

So if we now go back and look at the code, yes, this fails the pants on fire test that there's clearly something wrong with these numbers. What we were appending is walk of Homer, numTrials, 1. Well, numTrials is a constant. It's always 100. What I intended to write here was not numTrials but numSteps.

I actually did this the first time I wrote this simulation many years ago. I made this typo, if you will, and I got these bizarre answers. So I looked at the code and I said, well, that's actually wrong. No wonder it's always the same number. I'm calling it with a constant. The constant happens to be 100. So let's go fix the simulation. So this should have been numSteps.

Now let's run it again. Well, these results are pretty different. Now we see that in fact, they're increasing. Should I just look at this and be happy? Probably not. I should run my sanity check again and make sure I get the right results for zero, one, and two. So let's go back and do that, just to be a little bit safe.

So I'll just change this tuple of values to be-- and I should feel a lot better about this. The mean, the max, and the min are all zero when he doesn't take any steps. One is exactly what we should expect, and two is also-- the mean is where we would guess it to be, and the max is two, happened to take two steps in the same direction, and the min is zero, happened to end up where he started. So I've passed my sanity check. Doesn't mean my simulation is right, but at least I have reason to be hopeful. So getting back.

So we saw these results, and now we're getting the indication that in fact, contrary to what we might have thought, it does appear to be that the more steps the drunk takes, the further away the drunk ends up. And that was the usual drunk. We can try the masochistic drunk, and then we see something pretty interesting. I won't make you sit through it, but when we run it, here are the usual drunks, the numbers, and I just looked at it for 1,000 and 10,000 so it would fit on the screen. You see for the usual drunk, it's 26.8, roughly 90. Fair dispersion in the min and the max. And the masochistic drunk seems to be making considerably more progress than the usual drunk. So we see is this bias actually appears to be changing the distance.

Well, that's interesting. Now we could ask the question why? What's going on? And to do that, I want to go and start visualizing what's the trend? So rather than just looking at two numbers or three numbers, as we've been doing, I'm going to draw a pretty picture. Actually, I'm not going to draw. I'm a terrible artist. But Python will draw us some pretty pictures. We're going to simulate walks of multiple lengths for each kind of drunk, and then plot the distance at the end of each length walk for each kind of drunk.

I now digress for a moment to talk about how we do plotting. So we're going to use something called Pylab. I've listed here four really important libraries that you will surely end up using extensively if you continue to use Python for research purposes. NumPy adds vectors, matrices, and many high-level mathematical functions. Actually, it might be NumPy. It might be "Num-Pee." I'm not sure how to pronounce it. But we'll call it NumPy. So these are really useful things.

SciPy adds on top of that a bunch of mathematical classes and functions useful to scientists. Things like-- well, we'll look at some of them as we go on through the term. MatPlotLib adds an object-oriented programming interface for plotting. Anybody here used MATLAB? Great. Well you'll find that MatPlotLib is the Mat, think MATLAB. Lets you, in Python, use all the plotting stuff that you've come to either like or hate in MATLAB. So it's really convenient. If you know how to do plots in MATLAB, you'll know how to do it in Python. PyLab combines all of these to give you a MATLAB-like interface to Python. So once you have PyLab, you can do a lot of things that you would normally want to do in MATLAB, for example, produce weird-looking plots like this one.

I'm going to show you one of the many, many plotting commands. It is called plot. It takes two arguments, which must be sequences of the same length. The first argument is the x-coordinates, the second argument is the y-coordinates corresponding to the x ones. There are roughly three zillion optional arguments, and you'll see me use only a small subset of them. It plots the points in order. First the first xy, then the second xy, then the third xy.

Why is it important that I say it plots them in order? Because by default, as each point is plotted, it draws a line connecting one point to the next point to the next point. And so the order in which they're plotted will determine where the lines go. Now we'll see, as we go on, that we often don't draw the lines, but by default they are drawn.

Here's an example. You start by importing PyLab. Then I've given xVals and yVals1, and if I call pylab.plot of xVals, yVals1. Here is one of the arguments I can give it. I'm saying I'd like this to be plotted in blue, b for blue, and I'd like it to be plotted as a solid line, a single dash. And I want to give that line a label, which I've said is first.

YVals2 is a different list. I'll plot it again. Here I'm going to say I want a red dotted line, and the label will be second. And then after plotting it, I'm going to invoke pylab.legend, which puts this nice little box up here in the corner, in this case, saying that first is a solid blue line and second a dashed red line, I should say.

Now again, there are lots of arguments, lots of other arguments I could give in plot. Also legend, I can tell it where to put the legend, if I so choose. Here I've just said, put it wherever you happen to want to put it, or PyLab wants to put it. So a very simple way to produce a plot.

There are lots of details and many more examples about plotting in the assigned reading. We've posted a video that Professor Grimson produced for an online course, 600.1x. It's about a 50 minute video broken into multiple segments about how to use plotting in PyLab with a lot more detail than I've given you. You'll see if you read the code for this lecture. And as you see this lecture, there'll be lots of other plots showing up of different kinds. These are my two favorite online sites for finding out what to do. And of course, you can google all sorts of things.

That's all I'm going to tell you about how to produce plots in class, but we're going to expect you to learn a lot about it, because I think it's a really useful skill. And at the very least, you should feel comfortable that any plot I show you, you now-- obviously not right now-- but you will eventually know how to produce. So if you do these things, you'll be more than up to speed.

So I started by saying I wanted to plot the trends in distance, and they're interesting. So here's the usual drunk and the masochistic drunk. So you can see, sure enough, the usual drunk, this fuschia line is progressing very slowly and the masochistic drunk, considerably faster. I looked at these, and after looking at those two, I tried to figure out whether there was some mathematical explanation of what was going on, and decided, well, it looked to me like the usual drunk was moving at about the square root of the number of steps. Not so odd to think about it, if you go back to old Pythagoras here. And sure enough, when I plot, and I ran this simulation up to 100,000 steps. When I plot the square root of the number of steps, it's not identical, but it's pretty darn close. Seems to be moving just a tad faster than the square root, but not much. But who knows exactly? But pretty good.

And then the masochistic drunk seems to be moving at a rate of numSteps times 0.05. A less intuitive answer than the square root. Why do you think it might be doing that? Well, what we notice is that-- and we'll look at this-- maybe there's not much difference between what the masochistic drunk and the usual drunk do on the x-axis, east and west. In fact, they shouldn't be. But there should be a difference on the y-axis, because every time, 1/4 of the time, the drunk is taking a step north of 1.1 units, and 1/4 of the time, he's taking a step south of 0.9 units. And so 1/2 the time, the steps are diverging by a small fraction. And if we think about it, 0.1 1/2 the time. We divide it. We get 0.05. So at least we need to do some more analysis, but the data is pretty compelling here that it's a very good fit.

Well, let's look at the ending location. So here you see a rather different kind of plot. Here I'm showing that you can plot these things without connecting them by lines. And giving them different shapes. So what here I've said is that the masochistic drunk we're going to plot using red triangles, and the usual drunk I'm going to plot using black plus signs. And since I'm going to plot the location at the end of many walks, it doesn't make sense to draw lines connecting everything, because all we're caring about here is the endpoints. So since I only want the endpoints, I'm plotting them a different way.

So for example, I can write something like plot(xVals, yVals, and then if I do something like let's see. Just 'b0' in fact. What that says is plot blue circles. I could have written-- in fact I did write 'k+' and that says black plus signs. And I don't actually remember what I did to get the triangles, but it's in the code.

And so that's very flexible what you do. And as you can see here, we get the insight I'd communicated earlier that if you look at east and west, not much difference between this ball and that ball. They seem to be moving about the same spread, the same outliers. But this ball is displaced north. And not surprisingly, after 10,000 steps, you would not expect any of these points to be below zero, where you'd expect roughly half of these points to be below zero. And indeed that's about true.

And we see here what's going on that if we look at the mean absolute difference in x and y, we see that not a huge difference between the usual drunk and the masochistic drunk. There happens to be a distance. But a huge difference-- sorry, x and x. Comparing the two y values, there's a big difference, as you see here.

So what's the point of all this? It's not to learn about different kinds of drunks. It's to show how, by visualization, we can get insight into our data that if I just printed spreadsheets showing you all of these endpoints, it would be hard to make sense of what was there. So get accustomed to using plotting to help you understand data.

Now let's play a little bit more with the simulation. We looked at different kinds of drunks. Let's look at different kinds of fields. So I want to look at a field with what we'll call a wormhole in it. For those of you of a certain generation, you will recognize the Tardis. So the idea here is that the field is such that as you wander around it, everything is normal. But every once in a while you hit a place where you're magically transported to a different place. So it behaves very peculiarly.

So let's call this an OddField. Not odd numbers, but odd as in strange. So it's going to be a subclass of field. We're going to have a parameter that tells us how many worm holes it has. A default value of 1,000. And we'll see how we use xRange and yRange shortly.

So what are the first thing we do? Well, we'll call Field _init to initialize the field in the usual way. And then we're going to create a dictionary of wormholes. So for w in the range number of worm holes, I'm going to choose a random x and a random y in xRange minus xRange to plus xRange, minus yRange to yRange. So this is going to be where the worm holes are located.

And then for each of those, I'm going to get a random location where you're, in some sense, teleported to if you enter the wormhole. So here we're using random to get random integers. We've seen that before. And so the new location will be the location of the new x and the new y, and we're going to update this dictionary of wormholes to say that paired with the location x, y is newLoc.

Now when we move the drunk, and again this is just changing-- we're overriding moveDrunk, so we're overriding one of the methods in Field. So Field.moveDrunk will take a self and a drunk. It's going to get the x value, the y value, and if that is in the wormholes, it's going to move the drunk to the location associated with that wormhole. So we move a drunk, and if the drunk ends up being in the wormhole, he gets transported.

So we're using Field.moveDrunk. So notice that we're using the moveDrunk of the superclass, even though we're overriding it here. Because we've overridden it here, I have to say Field. to indicate I want the one from the superclass, not the subclass. And then we're doing something peculiar after the move.

So interestingly here, I've taken, I think, a usual drunk and plotted the usual drunk on a walk of 500 steps. One walk, and shown all the places the drunk visited. So we've seen three kinds of plots, one showing how far the drunk would get at different length walks, one showing all the places the drunk would end up with many walks of the same length, and here a single walk, all the places the drunk visits. And as you can see, the wormholes produce a profound effect, in this case, on where the drunks end up.

And again, you have the code. You can run this yourself and simulate it and see what you go.

And I think I've set random.Seed to zero in each of the simulations in the code, but you should play with it, change it, to just see that you'll actually get different results with different seeds.

Let me summarize here, and say the point of going through these random walks is not the simulations themselves, but how we built them. That we started by defining the classes. We then built functions corresponding to one trial, multiple trials, and reported the results. And then made a set of incremental changes to the simulation so that we could investigate different questions.

So we started with a simple simulation with just the usual drunk and the simple field, and we noticed it didn't work. How did we know it? Well, not because when we did the full simulation we had great insight. I probably could have fooled 1/2 of you and convinced you that that was a reasonable answer. But as soon as we went and did the sanity check, where we knew the answer, we could know something was wrong. And then we went and we fixed it. And then we went and we elaborated it at a step of a time. I first got a more sophistic-- I shouldn't say sophisticated. A different kind of drunk. And then we went to a different kind of field. Finally, we spent time showing how to use plots to get an insight. And in the remaining few minutes of the class, I want to go back and show you some of the plotting commands. To show you how these plots were produced.

So one of the things I did, since I knew I was going to be producing a lot of different plots, I decided I would actually not spend time worrying about what kind of markers-- those are the things like the triangles and the plus sign-- or what colors for each one individually, but instead I'd set up a styleIterator that would just return a bunch of different styles. So once and for all, I could define n styles, and then when I want to plot a new kind of drunk, I would just call the styleIterator to get the next style. So this is a fairly common kind of paradigm to say that I just want to do this once and for all. I don't want to have to go through each time I do this.

So what do the styles look like? Let me just get this window. Oh. So here it is. I said there were going to be three styles that I'm going to iterate through. Style one is going to be an m, I guess that's maroon with a line, a blue with a dashed line, and green with a line with a comma and a minus sign.

So these are called the styles. And you can control the marker, if you have a marker. You can control the line. You can control the color. Also you can control the size. You can give the sizes of all these things. What you'll see when you look at the code is I don't like the default styles things, because when they show up on the screen, they're too small.

So there's something called rcParams. Those of you who are Unix hackers can maybe guess where that name came from. And I've just said a bunch of things, like that my default line width will be four points. The size for the titles will be 20. You can put titles on the graphs. Various kinds of things. Again, once and for all trying to set some of these parameters so they get used over and over again.

And then finally down here, you'll see that I did things like you want to put titles on the slides. So on the graph. So here's the location at end of walk. Title is just a string. You want to label your x and y-axis, so I've labeled them here. And here I've said where I want the legend to appear in the lower center. I've also set the y-limits and the x-limits on the axis, because I wanted a little extra room. Otherwise, by default it will put points right on the axes, which I find hard to read.

Anyway, the point here is not that you understand all of this instantaneously. The point I want to communicate is that it's very flexible. And so if you decide you don't like the way a plot looks and you want to change it, and you know what you want it to look like, there's almost surely a way to make it do that. So don't despair. You can look at the references I gave earlier and figure that out.

Next lecture we're going to move on. No more random walks. We'll look at simulating other things, and in particular, we'll look at the question of how believable is a simulation? See you Wednesday if the world has not come to an end.