MIT OpenCourseWare
http://ocw.mit.edu

6.00 Introduction to Computer Science and Programming, Fall 2008

Please use the following citation format:

> Eric Grimson and John Guttag, *6.00 Introduction to Computer Science and Programming, Fall 2008*. (Massachusetts Institute of Technology: MIT OpenCourseWare). http://ocw.mit.edu (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit:
http://ocw.mit.edu/terms

MIT OpenCourseWare
http://ocw.mit.edu

6.00 Introduction to Computer Science and Programming, Fall 2008
Transcript – Lecture 24

OPERATOR: The following content is provided under a Creative Commons license. You're support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare course at ocw.mit.edu.

PROFESSOR: All right. Today's lecture, mostly I want to talk about what computer scientists do. We've sort of been teaching you computer science in the small, I want to pull back and think in the large about, what do people do once they learn about computer science? And then I'll wrap up with a quick overview of what I think we've accomplished this term.

So what does a computer scientist do? What they really do is, almost everything. Graduates of our department, other departments, have done things like animation for movies you've all seen, they keep airplanes from falling out of the sky, they help surgeons do a better job of brain surgery, that's something Professor Grimson has worked on. All sorts of exciting things. Mostly what they have in common, and it's incredibly varied, what you can do with computer science, but it all involves thinking computationally. And if I had to use a single phrase to describe 6.00, that's the phrase I would use. The semester is really about computational thinking. I think this will be a fundamental skill used by everyone in the world by the middle of the current century. And it'll be like the, it will become one of the three r's, reading writing arithmetic. And I haven't quite figured I how to put an r in the front of it, but sooner or later.

So what is it? Well, the process is the process that we've sort of been advocating throughout the semester. You identify or invent useful abstractions. Sometimes they're out there, and we can just pluck them. Sometimes you have to invent them ourselves. We then formulate a solution to a problem as some sort of a computational experiment. Typically using those abstractions. Then comes the part that many you get bogged down in, but it'll get easier for you as you go on. Is to design and construct a sufficiently efficient implementation of the experiment. And I want to emphasize the word sufficiently here, in that it only has to be fast enough to run it and get an answer. And it doesn't have to be the fastest possible. We then need to validate the experimental setup, convince ourselves that the code is right, and then run the experiment, and then evaluate the results, and then repeat as needed. And that's, of course, the key. By now you've all learned that these experiments rarely are done the first time you run them. But of course, this is true of experiments in biology and physics and chemistry and everything else. But this is the basic process. And I think, it's a view that's not universally held, but I think is the correct view of what computational thinking is all about. It's an experimental discipline.

So the two a's are abstraction, choosing the right abstraction, and operating in terms of multiple layers of abstraction simultaneously. And that's a little bit tricky. Something that you've worked on this term. You invent some beautiful, high level of abstraction. But then you have to think about it, and implement it using the rather low level abstractions provided by the programming language. So we'll do a simulation of, well, the stock market, we've looked at it just recently. And you've got this abstraction of a stock, and at one level, when we're thinking about the market, we think of a stock abstractly, as something that has a price that moves up and down. But then at the same time, we say, well, is the movement Gaussian, or uniform, and we're operating at a somewhat different level. And then at a lower level, we're asking questions like, well, how should we represent it? is? It a dictionary, is it a list, that sort of thing. And we always bounce back and forth and spend a lot of time thinking about how the layers relate.

Now the other thing that's important, and it really distinguishes, I think, computing from a lot of other disciplines. Is, we don't just write elegant descriptions of abstractions on paper, the way you say, would do in mathematics. But we have to then automate those attractions. And we always think in terms of, how can we mechanize the abstractions? And that's the computational part, in many ways. How can we do that? Well, we can do it because we have precise and exacting notations in which to express these models. So I'll say a few words about the model of stocks, or a few words about the model of a drunk wandering around a field, but I can't get away with just the words. Eventually I have to convert the words to code. And that's sort of the acid test of truth. Have I really understood what the words mean? Because if I don't, I can't actually convert it to code.

And that's both the cursing and the bless of being a computer scientist. It's the curse because, you can't fake it, right? And that's one of the reasons that many students end up spending a lot of time in a course like 6.00. Because they know whether or not their program works. When you write a proof in math, you can delude we into thinking it's a solid proof, when it isn't. When you write an essay, you can delude yourself into thinking it's brilliant, when it isn't. But here, you look at what it does, and it either did what you thought it would do or didn't. And so that means you sometimes have to work extra hard, but the blessing is, when it's done, you really know you've done it. And you've got something useful. And that's because there's a machine that will execute your descriptions. And that's what makes it fun.

So some examples of computational thinking. How difficult is this problem? That's what we talked about when we talked about complexity. And we talked about, for many problems, for all problems, there's an intrinsic difficulty of solving the problem with the computation. Independent of how efficient your particular solution is. And then the question is, how can I best solve it? Theoretical computer science has given us precise meaning to these and related questions. Thinking recursively, another good example. Where we take a seemingly difficult problem, and reformulate it into one which we know how to solve. Very often, it's a smaller instance of the original problem. And we say, gosh, this is hard to solve. But then we think, well suppose the list only had two elements in it, would I know how to sort it? Yeah. Well, then we can build up and say, therefore I can use that idea to sort a list of any size.

As part of this thinking recursively we learn about reduction, we learned how to, say, reduce the problem of deciding what courses to take to an optimization problem in terms of the knapsack problem. Once we did that, we could say, oh, we know how to solve all knapsack problems, dynamic programming has been around a long time, we'll just do that. So we do a lot of these reductions, in betting, transformations, transforming one problem into another, and simulations. So some other examples, choosing an appropriate representation, modeling the relevant abstracts of a problem to make it practical. And this is the essence of abstraction that computer scientists learn, is how to figure out what's relevant and ignore what is irrelevant, or less relevant. Focus on the important things. That's an enormous skill in not only computing, but in life.

We worry about worst-case scenarios, damage containment, error correction, all of this we talked about, we talked about debugging, defensive programming, making sure the types were right, but again these are very general things that we think about. And we think about the fact, sometimes, it's kind of a nice idea, the fact that there exists really hard problems is actually a good thing. Because that's what let's us do things like use encryption to get privacy of our data, or privacy for our communications, privacy of our phone calls. Because we have learned that some problems are intrinsically difficult to solve, and that's what coding is all about. Or encryption is all about. That's a nice thing.

I now want to talk more specifically about what one group of computer scientists does. And that's my research group. Which consists of mostly graduate students, but every year I try and incorporate a few undergraduates in it. We work in the area of medicine, it's also the area in which Professor Grimson does most of his work. The goal is pretty simple: we want to help people live longer and better lives. Right? I don't think any of you will think that's a worthless goal. But, a second goal is, we want to have fun while we're doing it. Because life is way too short to do anything that's not fun for very long. So we push the frontiers of computer science, electrical engineering, and medicine. Working in close collaboration with physicians, and here you've got logos of some of the hospitals with which we work. It is, of course, fairly technical work. We use machine learning, clustering, data mining, algorithm design, signal processing. And everything we do ends up, eventually, getting translated to code which we then run. And so we worry a lot about software systems and the quality. We do write systems that, for example, inject electrical signals into people's brains. And it's kind of important, when you do that, that the software does what it's intended to do. At least if you like the people.

So some specific activities, and this is only a sampling. We work on extracting clinically useful information from electrical signals in the human body. Mostly the heart, the brain, and connected anatomy. For those of you who've never studied anatomy, I've labeled the heart and the brain in the diagram for you. The two major projects in this area we've been working on, is predicting adverse cardiac events and detecting and responding to epileptic seizures. And I want to say a few words about each of those. We'll skip this. So example one, treating epilepsy. I suspect most of you are surprised by the fact that one percent of the world's population has epilepsy. This is true in almost every part of the world. It's true in the United States, it's true in underdeveloped countries in Central Africa. It seems to be one of the few diseases that's invariant to economic situations. One of the reasons that people are surprised by this number is, most folks who have epilepsy try to keep it a secret. There's a long history, getting back to witches, people with epilepsy were deemed to be witches, were burned at the stake, things of that nature. People who have epilepsy are not allowed by the FAA to work on maintenance of aircraft. It's a ridiculous restriction, but it's in the law, so I know someone with epilepsy who's kept it a secret because he doesn't want to lose his job. That's a different sermon.

It's characterized by recurrent seizures, generated by abnormal electrical activity in the brain. It's really less a disease than a description of symptoms, because there are lots of independent, different causes of it. It can be inherited, that's a significant fraction. Or it can be acquired, typically by some sort of insult to the brain. People who have strokes or serious infections will develop epilepsy. Hemorrhages, and increasingly head injuries. Turns out to be an illness that is of now great interested the Defense Department because people are coming back from Iraq with epilepsy. So those of you who don't have a strong stomach, you might want to avert your eyes. Here's one manifestation of it. On the left is an EEG, and that's, in this case, a recording of the electrical activity in the surface of this girl's brain. You can see she's wearing what looks like a very funny looking shower cap. That's where the electrodes are. And it's recording the electrical activity in the brain, on the right is obviously a video. Well, pretty frightening. Not all seizures are quite that frightening. In fact, most of them are not. Sometimes people have what are called absence seizures, where they just sort of black out for a period of time. Still not a good thing if you're, say, driving a car when that happens.

The key issue here, is that the onset of the seizure seems unpredictable. And in particular, it seems unpredictable to the person who has the seizure. And that's the real risk. Seizures are typically self-limiting. They will correct themselves over time. Eventually this girl will return to, quote, normal. The seizure itself will probably be over in a minute or two, and for about an hour she'll be confused, and then she'll be OK again. But, because they're unpredictable, even people who have as few as, say, two a year, it dominates their life. So, if you're that girl, and even if you're going to have one of those a year, you're never going to be allowed on a bicycle. Because if that happens when you're on a bicycle, it's really bad. If you're an adult, and you have one or two year, you should never drive a car. Imagine if you've been driving a car when that happened.

Almost everybody with epilepsy eventually suffers a serious injury. They have a seizure while they're on a flight of stairs, and they fall down and fracture their skull. They get intra-cranial hematomas, they get burns if they're cooking at the stove, they drown in the bathtub. And it's the unpredictability that's bad. Death is high, two to three times out of the general population. Typically by accident related to the seizure. And then there's something called sudden unexplained death in epilepsy patients, SUDEP, which is estimated to be one per 100 patients per year. Just will die, and no one knows why. Frequently it will be at night, while they're in bed, and the conjecture is, they have a seizure, and end up face down in the pillow, smother, that kind of thing. So you can imagine, if you're a parent, you have a child with this, you don't sleep much. Really very sad.

So we want to do something about that. And our idea is to detect the seizure early. There are two onset times, electrographic and clinical. Electrographic is when the brain first starts looking suspicious, and clinical is when there's a overt physical symptom, that stiffness you saw in that young girl. Probably it went by too fast for any of you to notice, but if you'd looked carefully at the EEG on the left of that picture, you would have seen it going abnormal before she had any clinical symptoms. Because the symptoms are caused by the electrical activity in the brain, by definition, it proceeds the symptoms, or at least doesn't follow them. So our idea was to try and detect the electrographic onset. If you could do this, you could provide warning. So you could tell somebody, sit down before you fall down. Get out of the bathtub. Back away from the stove. Just an oral warning would be tremendously useful. You could summon help. You could tell a parent, come quick, your child is having a seizure right now in bed or elsewhere.

There are fast-acting drugs, Ativan, which if you inhale at the start of a seizure, can stop it, reduce it. And our particular interest is neural stimulation. There is reason to believe, it's yet to be proved, that if you apply the correct electrical stimulation at the very beginning of a seizure, you can block it. Or at least ameliorate it greatly. So, here's a picture of an EEG. And I'll just point out, seizures can be tricky to find. You might think this is a seizure, but it isn't. That's blinking your eyes. Oh well. Here is the seizure, but in fact, it's begun about here. Where things don't look particularly ominous. So it's pretty subtle to detect. One of the problems is, EEG varies across patients. And epileptics have abnormal baselines. Yeah?

STUDENT: [INAUDIBLE]

PROFESSOR: Highly variable in patients, but that's kind of typical. But that would be plenty of time to back away from the stove, or get off a bicycle. Not so much time to get off the Mass Turnpike, if you're driving a car. But it would be great, if you can give 10 or 15 seconds warning. In fact, if you could give five seconds warning, it would be tremendously useful. And for the treatments, it's even different, right? Yes, you're exactly right, in that particular case. The problem is, one of the things I've discovered in medicine, my work, is that almost all healthy people look alike, and all sick people look different. So, EEG, if you have epilepsy, your quote, normal EEG, when you're not having a seizure, looks weird compared to people who don't have epilepsy. People have been building seizure detectors for about 40 years now, and they don't work very well. Every EEG that gets shipped comes with a seizure detector, and the first thing most hospitals do is, turn it off. Because it just doesn't work. The good news is, that even though each epileptics brain signals look different, once you've learned what an individual's signal looks like, it's pretty consistent. Both the interictal, between seizure, and the start of the seizure.

Well, that tells a computer scientist, let's use machine learning to figure out what that patient's EEG looks like. And build a highly specific detector that will work great for one person, and not at all for anybody else. And then we'll just install it, and you know, everyone gets their own. So instead of one size fits all, it's design a detector to fit the patient. We have been working on this now for about six years. We've done, we've looked at about 80 patients now, more. And, highly successful. And I'm not going to weigh you down with all the statistical tests and everything, but we've done the kind of things I talked about in class. Looking at the data, doing the statistics, and suffice it to say, it works great. For nine out of 10 patients, roughly. And for the tenth, we just have to say, sorry, we can't do it. But, you know, you can't help everybody all the time.

We're currently, what you see here, is a picture of a neural stimulator. The stimulator itself gets implanted under the clavicle, and then a wire runs up and wraps around the left vagus nerve, which is the longest nerve in the human body. It runs all away from the head, down into the lower intestine. Latin for wanderer. Does amazing things, like control whether you're hungry or not. Its main, one of its main functions, is it controls the parasympathetic nervous system for the heart. And it's the nerve that the brain uses to tell the heart, slow down. But, in addition to transmitting down from the brain, you can transmit up the nerve. And in fact, the left one, for reasons I don't understand, translates much better up than the right one does. Which transmits much better down. So use the left one. And the notion is, if at the start of a seizure, we put a current on that wire, we stimulate the brain and we stop the seizure. That's the theory. It's been shown to stop seizures in rats. But I don't much care about helping rats.

People purport to have shown it stops seizures in adults, or in people. The level of evidence is, honestly, ambiguous. Because what they've done is, they've implanted it, and they give the patient a magnet. And say, when you're having a seizure, swipe the magnet across, sorry about that, swipe the magnet across the stimulator, it will turn it on and stop your seizure. Well, could you imagine that little girl trying to swipe a magnet across her chest? Not likely. So most people don't do it successfully. Some people report, I thought I was about to have a seizure, I swiped the magnet and I didn't have it. Well, how do we know? Did they really stop it, or did they just imagine they were going to have one? So it's pretty ambiguous, but we're now in the process of doing some real tests at Beth Israel Deaconness Medical Center. And we've been admitting patients, and controlling the stimulator, and so far I'm optimistic.

All right, one more example. Predicting death. Well, that's kind of ominous sounding. About one and a quarter million Americans have an acute coronary syndrome each year. That's to say, some sort of a cardiac event. A heart attack, or an arrhythmia of some sort, various kinds of things, unstable angina. And of that one and a quarter million people, 15 percent to 20 percent of them will die of a cardiac-related event within the next 4 years. Pretty high. In fact, about 5% within the next 90 days, there about. So what do you do? The key thing, we have treatments. We actually know how to treat most of these things. But we don't know who to give the treatments to? So, for example, who gets an implanted defibrillator? Who should be treated aggressively with statins? We think we've found a new way to decide who should get which of these kinds of treatments. Something called morphological variability, which I'll talk about only briefly. We've tested it on a fairly large database. We have a database with about 8,000 patients. We've looked at 2 days for each patient, each day has 24 hours, each hour has 60 minutes, each minute has, on average, 70 heartbeats. That's a lot of heartbeats. Over a billion. So a lot of the techniques we've talked about this term, about how do you deal with big things, I live with every day. Actually, more accurately, my students live with every day, and I commiserate.

But the tests are pretty convincing. As an example, they're implanted defibrillators, they, too, go under the clavicle. Fortunately we have two clavicles. And then they connect to the heart, and they notice, they have a sensor, they notice the heart is beating very badly, and they shock the heart. So you've all seen medical shows, where people put paddles on, say, clear, the patient goes boom boom, and then they say, sinus rhythm restored. Well, this does the same thing, but you walk around with it. Now, you don't want to turn on all the time, because as you can see, it delivers quite a jolt. Interestingly, this is an article from this fall, from the New York Times, 90 percent of the people who get one implanted, and the device itself costs about $50,000, let alone the cost of implanting it. 90 percent receive negative medical benefit. How do I know that they received medical benefit? Well, I know that 90% of them are never turned on. Remember, there's a sensor that decides whether to apply the shock. 90 percent of them, the sensor never says, turn on. So 90 percent of them can deliver no benefit because they're never used. On the other hand, clearly having the surgery to implant this, is some risk. So that's why the benefit is not only neutral, but negative. You've had fairly serious surgery, you've risked infection, and, in fact, this year several people died. I think three or four died when leads broke off on these things.

So, clearly we're putting in too many of them, if only 10 percent are used. On the other hand, roughly every one minute and 50 seconds, somebody in this country dies a sudden cardiac death. More often once every two minutes. 100 of them during the course of this lecture. If these people all had defibrillators implanted, many of them would not die. So, we have a therapy, we just don't know who to give it to. So, what does this look like? Again, this is pretty strong stuff. That's what sudden cardiac death looks like. An apparently healthy guy, suddenly he's dead. It's things like this that remind us why we can do real useful things with our talents or skills. So how do you identify high risk cases? Lots of different ways, I won't go through them, but we're focusing again on the electrical signals in the heart. We're trying to evaluate the shape of the heartbeat and what's going on. I won't go through details, the only thing I wanted to point out is, we use dynamic programming to actually do this. So the key idea in our method depends upon dynamic programming.

We've evaluated it several times. Here's one evaluation. We took the 25 percent of the population that we thought was at the highest risk. And you'll notice that within, actually the first 30 days, 4.19 percent of those people were dead. This is a retrospective study, by the way. The ECG had been recorded, and we knew, we had follow-ups on them, and of the people we said were not, the other 3/4 of the population, a very small fraction died. So this is called a Kaplan Meier curve, and it shows a huge gap. And in fact, roughly speaking, if we think you were at high risk, you were 8 1/2 times more likely to die within a month, within 90 days. And the good news is, we don't do this just so we can say, make your will. Get your affairs in order. We actually do this so that we can say, all right, this is a person who should maybe get a defibrillator, maybe should have their arteries cleaned out. Maybe should take a blood thinner. Once we know who's at risk, we can do something about it. All right, all I want to do is just give you a flavor of the kinds of things you've been learning, are really useful, and can be used to really do things that are worth doing.

So wrapping up the term, what have you learned? Well, you know, I think we'll start with this. If you think what you could do in September, and what you can do now, you've really come a long way, right? So try and go back and do problem set number one, and see how long it takes you now, and how long it took you then. You'll discover that you really are at a place you were not in September.

We covered five major topics. Learning a language for expressing computations, that's Python. I already talked about the importance of having a notation. Learning about the process of writing and debugging programs. But more generally, learning about the process of moving from some relatively ambiguous problem statement. As you've all noticed in the current problem set, we left a lot of things unspecified. You know, and the email was flying back and forth, and often the answer was, you figure it out. Because that's one of the things we want you to learn. To go from some problem statement to a computational formulation. As a tool to that, you learned a basic set of recipes, bunch of different algorithms. And you learned how to use simulations to deal with problems that did not have nice closed form solutions. Why Python? Well, believe it or not, it's an easy language to learn, compared to other languages. Simple syntax, it's interpretive, which helps with the debugging, you don't have to worry about managing memory as you would in a language like C. It's modern. It supports, in a fairly elegant way, object-oriented programming and classes, and it's increasingly popular. It's used increasing in other subjects at MIT and other universities, increasing in industry, and a large and ever-growing set of libraries.

We talked about writing, testing, and debugging programs. And if I had one message I want you to remember about all of this, it's be slow. Slow down, take it one step at a time, take your time and be methodical, systematic, and careful. Understand the problem first. Then think about the overall structure of the solution, and the algorithms independently of how you're going to code it up. So you might say, well, I'm going to use dynamic programming. And that's an independent decision from Python or Java or C++ , or anything else to implement it. And in fact, it's pretty easy to translate a well-organized program from one language to another. Break it into small parts, and we talked about unit testing and the importance of that. Nobody can build anything big in one fell swoop. So, break it into small parts, identify useful abstractions, both functional and data, code and unit test each part, and first worry about functionality. Are you getting the correct answer? And then efficiency. And a piece of that is, learn to use pseudo code to help yourself out.

As I said earlier, be systematic when debugging. Think about the scientific method. It's not just something that we use to torture middle school students. And when you write a program and it doesn't work, ask yourself, why it did what it did? Focus on understanding why the program is doing what it's doing, rather than why it's not doing what you wanted it to? Because as soon as you know why it's doing what it's doing, you'll know how to fix it. But if you skip that step, you'll waste an awful lot of time. Going from problem statement to computation. We said a lot of this, you break the problem into a series of smaller problems. Ideally, reducing it to a problem you already have code to solve, or somebody else's code to solve. My first instinct when I have to solve a problem is, I type, I go to Google, and I type, I think about, oh this is really an x problem, a shortest path problem. I type Python shortest path, and I see what code pops up on my screen. But I can't do that until I've realized, oh, it's really a shortest path, or it's really a knapsack. If I type Python select courses, I won't get anything useful.

Think about what kind of output you'd like to see. I often start thinking about a program by first asking myself, what's the output going to look like? And I'll actually write the output down. And then I'll see if I can't write a program to generate that kind of output. Often you can take a problem formulated as an optimization problem, a lot of thing reduce to finding the minimum or maximum values, satisfying some set of constraints. So when you're given something like, how do I find the directions from here to Providence, think about it as an optimization problem. And also, think about whether you really have to solve it. It's very often the case that an approximate solution is all you really need. And they're a lot faster. So, we think about approximation formally, in the sense of, say, Newton Raphson, we looked at, successive approximation. But more informally, an approximation can be, solve a simpler problem. That you don't really need to solve the problem you thought needed to solve, if you solve something simpler, it will do just fine. So you don't find the best solution, you just find a pretty good solution. Or you simplify the assumptions and say, well, I'm going to find directions, and to make my problem a little simpler, I'm not going to allow myself to take any roads except highways to get from city a to city b. It may not be the shortest, but it's an easier problem to solve because are so many fewer roads.

We talked about algorithms, we talked about big o notation, orders of growth, that's very important. We talked a little bit about amortized analysis, that you might want to, say, sort something first if you were going to do a lot of searching. We looked at a bunch of kinds of algorithms. I've listed them here. I'm, by the way, going to send you a list with all this stuff on it, among other things, so don't feel you need to transcribe it. But a lot of, surprisingly large number of algorithms, to cover in one semester, when you think about it. Exhaustive enumeration, successive approximation, greedy, divide and conquer, decision trees, dynamic programming. Quite a lot. Specific algorithms and specific optimization problems. We spent a lot of time on simulation. It's important, but maybe not as important as all that time, relative to the semester, but it gave me an excuse to talk about some things that I wanted to cover. So it gave me a framework to talk a little bit about probability and statistics. And it's my view, personally, that every MIT student should be made to take such a course. Since they're not, I tried to sneak it in here. But it's all, it's important stuff to know.

It gave me an excuse to build some interesting programs. So, what I hope you noticed, is as I went through these things, I actually tried to develop the programs incrementally, to talk about the abstractions, to talk about how I would test them. And so, just as important as the simulation itself, was the process of building the simulation. And so I just shows that as a mechanism to go through some case studies of building some interesting programs. It let us talk about random choice. Much of the world is, or at least appears to be, stochastic. And therefore, we have to model randomness in programs. And even we don't have to, it can be used to solve problems that are not inherently random, like finding the value of pi. It let us look at the issue of assessing the quality of an answer. We look at things, said, do we believe that that answer is right or not? And building models of, at least parts of, the world. Some pervasive themes that we talked about. The power of abstraction, systematic problem solving, these will be useful even if you never again write a program in your whole life.

What next? Well, many of you, almost all of you, have worked very hard this semester. I hope you feel that you've got a return on your investment. Only you know that. As I said earlier, take a look at the problem sets, and see whether you think you've learned anything. I'd like you to remember after you leave this course, that you've got a skill you can use to solve problems. And when you sit down, and there's some issue that you don't know, just knock out a program to get your answer. I do it all the time, and it's very useful.

If you want to take more courses, there are others you can take. 6.01, the first course for Course 6 majors, you will find easy. Experience says, people who take 6.00 first don't understand what all the fuss is about 6.01. You'll also find it fun and interesting. 6.034, the introduction to AI, is now using Python. You know everything you need to know to take that. 6.005, which is software engineering, you know everything you need to take that. And, for many of you, you can take 6.006, which is a really interesting kind of algorithms course. More advanced algorithms, particularly those you who like math would find 6.006 really fun. So if you want to take more courses, there are courses to take. Thank you all, good luck with the rest of the semester.