MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.081—Introduction to EECS I
Spring Semester, 2007

Lecture 1 Notes

Computing and the world

When the field of Computer Science was born, the majority of applications that motivated computer scientists were "self-contained," in the sense that they operated on data that had already been gathered and tidied into a database or other regularized format. Such systems include billing and inventory management, hospital records, etc.

The data were assumed to be correct and discrete. The foundations of computer science were logical: the same formalism was useful for describing the operation of a computer as well as the meaning of the data on which the computer was operating. There was no real modeling of uncertainty.

Most programs had no notion of control; their actions consisted of printing out "answers" to static questions. More recently, UI programming has forced CS to think about input/output loops, but the inputs are typically keystrokes and mouse-clicks and the outputs updates to an array of pixels on the screen.

Most of modern CS theory and software engineering has been derived from this view.

As computers become more ubiquitous and more connected to each other and to a complex environment, we need to take a much different view, which is closer to the view that electrical engineers have traditionally taken. Systems will have to be fundamentally adaptive, because we cannot know the details of the environments in which they will be deployed. Data of all kinds (audio, video, sensor readings, natural language text) from noisy, distributed, and unreliable sources will have to be processed and fused. Systems will have take the idea of "control" much more seriously: which packets to send where, when? what data to display to the user? where to deploy a set of autonomous vehicles?

The domains in which these systems will have to operate will be incompletely understood, ill-specified, and dynamic. It will be impossible to write a specification for the correctness of the computer system, in the old sense. The worst case in these domains is death or disaster, and we won't be able to guarantee success; instead, we will need to formulate new models of success that involve trade-offs of many different quantities (time, energy, death, etc.)

Systems of this kind can be and have been built using the current models, but they are brittle, time-consuming, and error prone. We need to build adaptivity and models of dynamics and uncertainty into the the very foundations of our theory and practice. This course is a first step in that direction.

Modularity, Abstraction, Modeling

The job of an engineer is to build highly complex systems. But there is only so much complexity a single person can hold in their head at a time. So how can one person or small team of people build a highly complex system? Through the power of modularity and abstraction.

Modularity allows an engineer to build components that can be re-used.

Abstraction allows an engineer, after constructing a module (using software programs or wires or gears), to forget the details of its construction and remember a simpler description of how it will interact with other modules (it computes the square root, or doubles the voltage, or changes the direction of motion).

Now, the engineer can move up a level of abstraction and construct a new module by putting together several previously-built modules, thinking only of their abstract descriptions, and not their implementations. This process gives the engineer the ability to construct systems with complexity far beyond their individual understanding.

Any module can be described in a large number of ways. We might describe the circuitry in a digital watch in terms of how it behaves as a clock and a stopwatch, or in terms of voltages and currents within the circuit, or in terms of the heat produced at different parts of the circuitry. Each of these is a different *model* of the watch. Different models will be appropriate for different tasks: there is no single correct model.

The primary theme of this course will be to learn about different methods for building modules out of primitives, and of building different abstract models of them, so that we can analyze or predict their behavior, and so we can recombine them into even more complex systems. The same fundamental principles will apply to software, to control systems, and to circuits.

Models of computation

Our first domain for studying modularity and abstraction will be software. There are lots of different ways of thinking about modularity and abstraction for software. Different models will make some things easier to say and do and others more difficult, making different models appropriate for different tasks.

Inside the computer

Most of the models in computer science are about computing some sort of function: given an input, compute an output. Examples include calculating the stresses and strains on an architectural design or deciding what set of web pages best matches a query given by a user.

Imperative computation Most of you have probably been exposed to an imperative model of computer programming, in which we think of programs as giving a sequential set of instructions to the computer to *do* something. And, in fact, that is how the internals of the processors of computers are typically structured. So, in Java or C or C++, you write typically procedures that consist of lists of instructions to the computer:

- Put this value in this variable
- Square the variable
- Divide it by pi
- If the result is greater than 1, return the result

In this model of computation, the primitive computational elements are basic arithmetic operations and assignment statements. We can combine the elements using: sequences of statements, and control structures such as `if`, `for`, and `while`. We can abstract away from the details of a computation

by defining a procedure that does it. Now, the engineer only needs to know the specifications of the procedure, but not the implementation details, in order to use it.

Functional computation Another style of programming is the functional style. In this model, we gain power through function calls. Rather than telling the computer to do things, we ask it questions: What is 4 + 5? What is the square root of 6? What is the largest element of the list?

These questions can all be expressed as asking for the value of a function applied to some arguments. But where do the functions come from? The answer is, from other functions. We start with some set of basic functions (like "plus"), and use them to construct more complex functions.

This method wouldn't be powerful without the mechanisms of conditional evaluation and recursion. Conditional functions ask one question under some conditions and another question under other conditions. Recursion is a mechanism that lets the definition of a function refer to the function being defined. Recursion is as powerful as iteration.

In this model of computation, the primitive computational elements are typically basic arithmetic and list operations. We combine elements using function composition (using the output of one function as the input to another), `if`, and recursion. We use function definition as a method of abstraction, and the idea of higher-order functions (passing functions as arguments to other functions) as a way of capturing common high-level patterns.

Data structures In either style of asking the computer to do work for us, we have another kind of modularity and abstraction, which is centered around the organization of data.

At the most primitive level, computers operate on collections of (usually 32 or 64) bits. We can interpret such a collection of bits as representing different things: a positive integer, a signed integer, a floating-point number, a boolean value (true or false), one or more characters, or an address of some other data in the memory of the computer. Python gives us the ability to use all of these primitives, except for addresses directly.

There is only so much you can do with a single number, though. We'd like to build computer programs that operate on representations of documents or maps or circuits or social networks. To do so, we need to aggregate primitive data elements into more complex *data structures*. These can include lists, arrays, dictionaries, and other structures of our own devising.

Here, again, we gain the power of abstraction. We can write programs that do operations on a data structure representing a social network, for example, without having to worry about the details of how the social network is represented in terms of bits in the machine.

Object-oriented programming: computation + data structures Object-oriented programming is a style that applies the ideas of modularity and abstraction to execution and data at the same time.

An *object* is a data structure, together with a set of procedures that operate on the data. Basic procedures can be written in an imperative or a functional style, but ultimately there is imperative assignment to state variables in the object.

One major new type of abstraction in OO programming is "generic" programming. It might be that all objects have a procedure called `print` associated with them. So, we can ask any object to print itself, without having to know how it is implemented. Or, in a graphics system, we might

have a variety of different objects that know their x,y positions on the screen. So each of them can be asked, in the same way, to say what their position is, even though they might be represented very differently inside the objects.

In addition, most object-oriented system support inheritance, which is a way to make new kinds of objects by saying that they are mostly like another kind of object, but with some exceptions. This is another way to take advantage of abstraction.

Anti-dogma Arguments about the choice of programming paradigms and language often take on a religious tone, with proponents of each as the one true way. Generally speaking, we advocate methods with simple, clear semantics, and picking the right tool for the job. How do you know what tool is right? Whichever one will end up with the simplest, clearest program. In the end, of course, there's some element of taste in deciding what is simple and clear; but that's life.

Interaction with the world

Increasingly, computer systems need to interact with the world around them, receiving information about the external world, and taking actions to affect the world. Furthermore, the world is dynamic. As the system is computing, the world is changing. This description applies to:

- User interfaces, where the input comes from user actions (keystrokes, mouse clicks, speech, gesture) and the output is typically the display of information, via computer monitor, voice-mail message, SMS, etc.;
- Cars, where the input comes from temperature and pressure sensors in the engine and the output controls various valves; or
- Robots, where the input comes from cameras and distance sensors and the output is voltages to motors in the wheels and/or arm.

There are a variety of different ways for organizing computations that interact with an external world. Generally speaking, such a computation needs to:

- get information from sensors
- perform computation, remembering some of the results
- take actions to change the outside world

But these operations can be put together in different styles.

Imperative The most immediately straightforward style for constructing a program that interacts with the world is the basic imperative style. A library of special procedures is defined, some of which read information from the sensors and others of which cause actions to be performed. Example procedures might move a robot forward a fixed distance, or send a file out over the network, or move video-game characters on the screen.

In this model, we could naturally write a program that moves an idealized robot in a square, if there is space in front of it.

```
if sonarSensor(1) > 1:
    move(1)
    turn(90)
    move(1)
    turn(90)
    move(1)
    turn(90)
    move(1)
    turn(90)
```

The major problem with this style of programming is that the programmer has to remember to check the sensors sufficiently frequently. For instance, if the robot checks for free space in front, and then starts moving, it might turn out that a subsequent sensor reading will show that there is something in front of it, either because someone walked in front of the robot or because the previous reading was erroneous. It is hard to have the discipline, as a programmer, to remember to keep checking the sensor conditions frequently enough, and the resulting programs can be quite difficult to read and understand.

Transducer An alternative view is that programming a system that interacts with an external world is like building a *transducer* with internal state. (Add figure). Think of a transducer as a processing box that runs continuously. At fixed time steps (think of this as happening many times per second), the transducer reads all of the sensors, does a small amount of computation, stores some values it will need for the next computation, and then generates output values for the actions.

This computation happens over and over and over again. Complex behavior can arise out of the temporal pattern of inputs and outputs. So, for example, a robot might try to move forward without hitting something by doing:

```
distToFront = readFrontSonar()
setForwardMotorSpeed(0.7 * (distToFront - 0.1))
```

Executed repeatedly, this program will automatically modulate the robot's speed to be proportional to the free space in front of it.

We could make it average over recent sensor values, to smooth out sensor noise, by doing something like:

```
distToFront = readFrontSonar()
avgDistToFront = 0.5 * avgDistToFront + 0.5 * distToFront
setForwardMotorSpeed(0.7 * (avgDistToFront - 0.1))
```

We'd have to initialize `avgDistToFront` to some value (maybe 0) before this whole process started.

The main problem with the transducer approach is that it can be difficult to do tasks that are fundamentally sequential (like the example of driving in a square, shown above).

Interrupt-driven User-interface programs are often best organized differently, as *event-driven* (also, interrupt driven) programs. In this case, the program is specified as a collection of functions (possibly with side effects that change some global state) that are attached to particular events that can take place. So, for example, there might be functions that are called when the mouse is clicked on each button in the interface, when the user types into a text field, or when the temperature of a reactor gets too high. An "event loop" runs continuously, checking to see if any of triggering events happens, and, if it does, calls the associated function.

As the number and frequency of the conditions that need responses increases, it can be difficult to keep a program like this running well, and to guarantee a minimum response time to any event.

Parallel processes Another strategy is to organize a system as a collection of smaller loops (called processes) that are all being executed simultanously by the computer. The fact is that computers can't (typically) execute multiple processes in parallel, but they can execute the commands of multiple processes in an interleaved fashion, making it seem as if they are happening simultaneously. A robot might have parallel processes for looking for trash to pick up and for making sure it doesn't run into walls.

This approach, when everything is working, can be compact and beautiful. But it can be very hard to debug: every time the program is run, the processes will be interleaved a little bit differently, and so it can often happen that a program will have an error in it that only shows up in 1 out of 100 runs, making it very hard to find.

This class In our work in this class, we will typically use the transducer organization for our programs, largely because it is the simplest to debug.

Within such a program, we have two function-computation problems:

- How to change our memory on this step, based on the old memory and the new sensory data; and
- What output to generate, based on the old memory and the new sensory data.

So, for instance, in the program that computed an average of the sonar readings above, the memory of the program is stored in the variable `avgDistToFront`. On each step, that memory value is updated based on the sensory data from `readFrontSonar()`, and it is used to generate an output command to the motors, via `setForwardMotorSpeed`.

Digital circuits and control systems can be modeled using essentially the same abstraction.

New in Python

We assume you know how to program in some language, but are new to Python. We'll use Java as an informal running comparative example.

You should read through the Python tutorial to get a basic grounding in the language. Here are what we think are the most important differences between Python and what you already know about programming.

Shell

Python is designed to be easy for a user to interact with. It comes with an interactive mode called a *listener* or *shell*. The shell gives a prompt (usually something like `>>>`) and waits for you to type in a Python expression or program. Then it will evaluate the expression you typed in and print out the value of the result. So, for example, an interaction with the Python shell might look like this:

```
>>> 5 + 5
10
>>> x = 6
>>> x
6
>>> x + x
12
>>> y = 'hi'
>>> y + y
'hihi'
>>>
```

So, you can use Python as a fancy calculator. And as you define your own procedures in Python, you can use the shell to test them or use them to compute useful results.

Indentation and line breaks

Every programming language has to have some method for indicating grouping. Here's how you write an if-then-else structure in Java:

```
    if (s == 1){
        s = s + 1;
        a = a - 10;
    } else {
        s = s + 10;
        a = a + 10;
    }
```

The braces specify what statements are executed in the `if` case. It is considered good style to indent your code to agree with the brace structure, but it isn't required. In addition, the semi-colons are used to indicate the end of a statement, independent of where the line breaks in the file are. So, the following code fragment has the same meaning as the previous one.

```
    if (s == 1){
s = s
+ 1;     a = a - 10;
    } else {
                s = s + 10;
    a = a + 10;
    }
```

In Python, on the other hand, there are no braces for grouping or semicolons for termination. Indentation indicates grouping and line breaks indicate statement termination. So, in Python, we'd write the previous example as

```
if s == 1:
    s = s + 1
    a = a - 10
else:
    s = s + 10
    a = a + 10
```

There is no way to put more than one statement on a single line. If you have a statement that's too long for a line, you can signal it with a backslash:

```
aReallyLongVariableNameThatMakesMyLinesLong = \
        aReallyLongVariableNameThatMakesMyLinesLong + 1
```

Is one method better than the other? No. It's entirely a matter of taste. But if you're going to use Python, you need to remember about indentation and line breaks being significant.

Types and declarations

Java programs are what is known as *statically and strongly typed*. That means that the types of all the variables must be known at the time that the program is written. This means that variables have to be declared to have a particular type before they're used. It also means that the variables can't be used in a way that is inconsistent with their type. So, for instance, you'd declare `x` to be an integer by saying

```
int x;
x = 6 * 7;
```

But you'd get into trouble if you left out the declaration, or did

```
int x;
x = "thing";
```

because a *type checker* is run on your program to make sure that you don't try to use a variable in a way that is inconsistent with its declaration.

In Python, however, things are a lot more flexible. There are no variable declarations, and the same variable can be used at different points in your program to hold data objects of different types. So, this is fine, in Python:

```
if x == 1:
    x = 89.3
else:
    x = "thing"
```

The advantage of having type declarations and compile-time type checking, as in Java, is that a compiler can generate an executable version of your program that runs very quickly, because it can be sure what kind of data is stored in each variable, and doesn't have to check it at runtime. An additional advantage is that many programming mistakes can be caught at compile time, rather than waiting until the program is being run. Java would complain even before your program started to run that it couldn't evaluate

```
3 + "hi"
```

Python wouldn't complain until it was running the program and got to that point.

The advantage of the Python approach is that programs are shorter and cleaner looking, and possibly easier to write. The flexibility is often useful: In Python, it's easy to make a list or array with objects of different types stored in it. In Java, it can be done, but it's trickier. The disadvantage of the Python approach is that programs tend to be slower. Also, the rigor of compile-time type checking may reduce bugs, especially in large programs.

Procedures

In Python, the fundamental abstraction of a computation is as a procedure (we will often slip and call them "functions" instead). A procedure that takes a number as an argument and returns the argument value plus 1 is defined as:

```
def f(x):
    return x + 1
```

The indentation is important here, too. All of the statements of the procedure have to be indented one level below the `def`. It is crucial to remember the `return` statement at the end, if you want your procedure to return a value. So, if you defined `f` as above, then played with it in the shell,[1] you might get something like this:

```
>>> f
<function f at 0x82570>
>>> f(4)
5
>>> f(f(f(4)))
7
>>>
```

If we just evaluate `f`, Python tells us it's a function. Then we can apply it to 4 and get 5, or apply it multiple times, as shown.

What if we define

[1] Although you can type procedure definitions directly into the shell, you won't want to work that way, because if there's a mistake in it, you'll have to type the whole thing in again. Instead, you should type your procedure definitions into a file, and then get Python to evaluate them. Look at the documentation for the programming environment we're using for an explanation of how to do that.

```
def g(x):
    x + 1
```

Now, when we play with it, we might get something like this:

```
>>> g(4)
>>> g(g(4))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in g
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>>
```

What happened!! First, when we evaluated `g(4)`, we got nothing at all, because our definition of `g` didn't return anything. Well...strictly speaking, it returned a special value called `None`, which the shell doesn't bother printing out. The value `None` has a special type, called `NoneType`. So, then, when we tried to apply `g` to the result of `g(4)`, it ended up trying to evaluate `g(None)`, which made it try to evaluate `None + 1`, which made it complain that it didn't know how to add something of type `NoneType` and something of type `int`.

Whenever you ask Python to do something it can't do, it will complain. You should learn to read the error messages, because they will give you valuable information about what's wrong with what you were asking for.

Lists

Python has a built-in list data structure that is easy to use and incredibly convenient. So, for instance, you can say

```
>>> y = [1, 2, 3]
>>> y[0]
1
>>> y[2]
3
>>> len(y)
3
>>> y.append(4)
>>> y
[1, 2, 3, 4]
>>> y[1] = 100
>>> y
[1, 100, 3, 4]
>>>
```

A list is written using square brackets, with entries separated by commas. You can get elements out by specifying the index of the element you want in square brackets, but *note that the indexing starts with 0*!

You can add elements using `append` (and in other ways, which we'll explore in the exercises). The syntax that is used to append an item to a list, `y.append(4)`, is a little bit different than anything we've seen before. It's an example of Python's *object-oriented programming* facilities, which we'll study in detail later. Roughly, you can think of it ask asking the object `y` for its idea of how to append an item, and then applying it to the integer 4.

You can change an element of a list by assigning to it: on the left-hand side of the assignment statement is `y[1]`. Something funny is going on here, because if it were on the right-hand side of an assignment statement, the expression `y[1]` would have the value 2. But it means something different on the left-hand side: it names a place where we can store a value (just as using the name of a variable `x` on the left-hand side is different from using it on the right-hand side). So, `y[1] = 100` changes the value of the second element of `y` to be 100.

In Python, you can also make something like a list, but called a *tuple*, using round parentheses instead of square ones: `(1, 2, 3)`. Tuples cannot have their elements changed, and also cause some syntactic confusion, so we'll mostly stick with lists.

Iteration over lists

What if you had a list of integers, and you wanted to add them up and return the sum? Here are a number of different ways of doing it.[2]

First, here is something like you might have learned to write in a Java class (actually, you would have used `for`, but Python doesn't have a `for` that works like the one in C and Java).

```
def addList1(l):
    sum = 0
    listLength = len(l)
    i = 0
    while (i < listLength):
        sum = sum + l[i]
        i = i + 1
    return sum
```

It increments the index `i` from 0 through the length of the list - 1, and adds the appropriate element of the list into the sum. This is perfectly correct, but pretty verbose and easy to get wrong.

```
def addList2(l):
    sum = 0
    for i in range(len(l)):
        sum = sum + l[i]
    return sum
```

[2]For any program you'll ever need to write, there will be a huge number of different ways of doing it. How should you choose among them? The most important thing is that the program you write be correct, and so you should choose the approach that will get you to a correct program in the shortest amount of time. That argues for writing it in the way that is cleanest, clearest, shortest. Another benefit of writing code that is clean, clear and short is that you will be better able to understand it when you come back to it in a week or a month or a year, and that other people will also be better able to understand it. Sometimes, you'll have to worry about writing a version of a program that runs very quickly, and it might turn out that in order to make that happen, you'll have to write it less cleanly or clearly or briefly. But it's important to have a version that's correct before you worry about getting one that's fast.

Here's a version using Python's `for` loop. The `range` function returns a list of integers going from 0 to up to, but not including, its argument. So `range(3)` returns `(0, 1, 2)`. A loop of the form

```
for x in l:
   something
```

will be executed once for each element in the list `l`, with the variable `x` containing each successive element in `l` on each iteration. So,

```
for x in range(3):
    print x
```

will print `1 2 3`. Back to `addList2`, we see that `i` will take on values from 0 to the length of the list minus 1, and on each iteration, it will add the appropriate element from `l` into the sum. This is more compact and easier to get right than the first version, but still not the best we can do!

```
def addList3(l):
    sum = 0
    for v in l:
        sum = sum + v
    return sum
```

This one is even more direct. We don't ever really need to work with the indices. Here, the variable `v` takes on each successive value in `l`, and those values are accumulated into `sum`.

For the truly lazy, it turns out that the function we need is already built into Python. It's called `sum`:

```
def addList4(l):
    return sum(l)
```

Now, what if we wanted to increment each item in the list by 1? It might be tempting to take an approach like the one in `addList3`, because it was so nice not to have to mess around with indices. Unfortunately, if we want to actually change the elements of a list, we have to name them explicitly on the left-hand side of an assignment statement, and for that we need to index them. So, to increment every element, we need to do something like this:

```
def incrementElements(l):
    for i in range(len(l)):
        l[i] = l[i] + 1
```

We didn't return anything from this procedure. Was that a mistake? What would happen if you evaluated the following three expressions in the Python shell?

```
>>> y = [1, 4, 6]
>>> incrementElements(y)
>>> y
```

Next week, when we look at higher-order functions, we'll see another way to do both `addList` and `incrementElements`, which many people find more beautiful than the methods shown here.

Modules

As you start to write bigger programs, you'll want to keep the procedure definitions in multiple files, grouped together according to what they do. So, for example, I might package a set of utility functions together into a single file, called `utility.py`. This file is called a `module` in Python.

Now, if I want to use those procedures in another file, or from the the Python shell, I'll need to say

```
import utility
```

Now, if I have a procedure that file called `foo`, I can use it in this program with the name `utility.foo`. You can read more about modules in the Python documentation.

Interaction and Debugging

We encourage you to adopt an interactive style of programming and debugging. Use the Python shell a lot. Write small pieces of code and test them. It's much easier to test the individual pieces as you go, rather than to spend hours writing a big program, and then find it doesn't work, and have to sift through all your code, trying to find the bugs.

But, if you find yourself in the (inevitable) position of having a big program with a bug in it, don't despair. Debugging a program doesn't require brilliance or creativity or much in the way of insight. What it requires is persistence and a systematic approach.

First of all, have a test case (a set of inputs to the procedure you're trying to debug) and know what the answer is supposed to be. To test a program, you might start with some special cases: what if the argument is 0 or the empty list? Those cases might be easier to sort through first (and are also cases that can be easy to get wrong). Then try more general cases.

Now, if your program gets your test case wrong, what should you do? Resist the temptation to start changing your program around, just to see if that will fix the problem. Don't change any code until you know what's wrong with what you're doing now, and therefore know that the change you make is going to correct the problem.

Ultimately, for debugging big programs, it's most useful to use a software development environment with a serious debugger. But these tools can sometimes have a steep learning curve, so in this class we'll learn to debug systematically using "print" statements.

Start putting print statements at all the interesting spots in your program. Print out intermediate results. Know what the answers are supposed to be, for your test case. Run your test case, and find the first place that something goes wrong. You've narrowed in on the problem. Study that part of the code and see if you can see what's wrong. If not, add some more print statements, and run it again. **Don't try to be smart....be systematic and indefatigable!**