MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Course notes for Week 1

There will be three handouts issued in 6.01 each week:

1. *Weekly assignment:* This includes work to do in the software lab and the design lab, as well as a two-part homework—one part due before the design lab, one part due before the following week's lecture.

2. *Weekly course notes:* (This handout is the course notes for week 1.) There is no course textbook. Instead, we'll distribute notes that cover the course material as we progress though it week by week. This week there are two distinct sets of note (all in this document: a high-level overview of the goals, approach, and material of the course, and detailed technical notes on Python).

3. *Weekly lecture handout:* This is to help you follow along in lecture and to review afterwards.

All three handouts will be distributed in lecture and also available online from the 6.01 web site `mit.edu/6.01`, linked from the course calendar. You should bring all three handouts with you to the software and design labs.

Course Overview

1 Goals for 6.01

We have a number of goals for this course. Our primary goal is to enhance your ability to solve complex problems by strengthening your skill in the most pervasive strategy for dealing with complexity: using abstraction and modularity. We will examine the use of abstraction and modularity in a number of contexts associated with problems in electrical engineering (EE) and computer science (CS), as we hope to help you develop a more fundamental understanding of abstraction and modularity. We also hope that you will see more than just the similarities in broad perspective, but will also develop skills in using several specific abstraction strategies that have persisted in importance. To accomplish our primary goal, we will study how to analyze and design systems that interact with, and attempt to control, an external environment. Such systems include everything from low-level controllers like heat regulators or cardiac pacemakers, to medium-level systems like automated navigation or virtual surgery, to high-level systems that provide more natural human-computer interfaces.

Our second goal is for you to learn that making mathematical models of real systems can help in the design and analysis of those systems; and to give you practice with building those models. In particular, we hope you will develop insight in to the difficult step of deciding which aspects of the real world are important to the problem being solved, and then how to model in ways that give insight into the problem.

We also hope to engage you more actively in the educational process. Most of the work of this course will not be like typical problems from the end of a chapter. You will work individually and in pairs to solve problems that are deeper and more open-ended. There will not be a unique right answer. Argument, explanation, and justification of approach will be more important than the answer. We hope to expose you to the ubiquity of trade-offs in engineering design. It is rare that a single approach will be best in every dimension; some will excel in one way, others in a different way. Deciding how to make such trade-offs is a crucial part of engineering.

Another way in which we hope to engage you in the material is by having many of you return to the course as a lab assistant. Having a large number of lab assistants in the class means that you can be given more interesting open-ended design problems, because there will be staff readily available to make sure you do not get stuck. Even more important, the lab assistants will question you as you go; to challenge your understanding and help you see and evaluate a variety of approaches. This Socratic method has proven to be of great intellectual value to both classroom students and lab assistants.

Finally, of course, we have the more typical goals of teaching exciting and important basic material from electrical engineering and computer science, including modern software engineering, linear systems analysis, electronic circuits, and estimation and decision-making. This material all has an internal elegance and beauty, as well as crucial role in building modern EE and CS systems.

2 Modularity, abstraction, and modeling

Whether proving a theorem by building up from lemmas to basic theorems to more specialized results, or designing a circuit by building up from components to modules to complex processors, or designing a software system by building up from generic procedures to classes to class libraries, humans deal with complexity by exploiting the power of abstraction and modularity. And this is because there is only so much complexity a single person can hold in their head at a time.

Modularity is the idea of building components that can be re-used; and abstraction is the idea that after constructing a module (be it software or circuits or gears), most of the details of the module construction can be ignored and a simpler description used for module interaction (the module computes the square root, or doubles the voltage, or changes the direction of motion).

One 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 one the ability to construct systems with complexity far beyond what would be possible if it were necessary to understand each component in detail.

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.

2.1 Example problem

Imagine that you needed to make a robot that would roll up close to a light bulb and stop a fixed distance from it. The first question is, how can we get electrical signals to relate to the physical phenomena of light readings and robot wheel rotations? There is a whole part of electrical engineering related to the design of physical devices that connect to the physical world in such a way that some electrical property of the device relates to a physical process in the world. For example, a light-sensitive resistor (photo-resistor) is a sensor whose resistance changes depending on light intensity; a motor is an effector whose rotor speed is related to the voltage across its two terminals. In this course, we will not examine the detailed physics of sensors and effectors, but will concentrate on ways of designing systems that use sensors and effectors to perform both simple and more complicated tasks. To get a robot to stop in front of a light bulb, the problem will be to find a way to connect the photo-resistor resistor to the motor, so that the robot will stop at an appropriate distance from the bulb.

2.2 An abstraction hierarchy of mechanisms

Given the light-sensitive resistor and the motor, we could find all sorts of ways of connecting them, using bits of metal and ceramic of different kinds, or making some kind of magnetic or mechanical linkages. The space of possible designs of machines is enormous.

One of the most important things that engineers do, when faced with a set of design problems, is to standardize on a *basis set* of components to use to build their systems. There are many reasons for standardizing on a basis set of components, mostly having to do with efficiency of understanding and of manufacturing. It's important, as a designer, to develop a repertoire of standard bits and

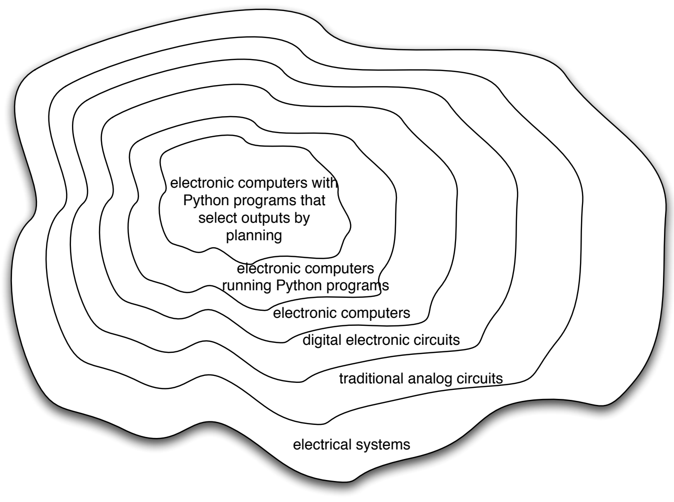

electronic computers with Python programs that select outputs by planning
electronic computers running Python programs
electronic computers
digital electronic circuits
traditional analog circuits
electrical systems

Figure 1: Increasingly constrained systems.

pieces of designs that you understand well and can put together in various ways to make more complex systems. If you use the same basis set of components as other designers, you can learn the techniques from them, rather than having to re-invent them yourself. And other people will be able to readily understand and modify your designs.

We can often make a design job easier by limiting the space of possible designs, and by standardizing on:

- a basis set of *primitive* components;
- ways of *combining* the primitive components to make more complex systems; and
- ways of “packaging” or *abstracting* pieces of a design so they can be reused.

Very complicated design problems can become tractable using such a *primitive-combination-abstraction* (PCA) approach. In this class, we will examine and learn to use a variety of of PCA strategies common in EE and CS, and will even design some of our own, for special purposes. In the rest of this section, we will hint at some of the PCA systems we’ll be developing in much greater depth throughout the class. Figure 1 shows one view of this development, as a successive set of restrictions of the design space of mechanisms.

One very important thing about abstract models is that once we have fixed the abstraction, it will usually be possible for it to be implemented using a variety of different underlying substrates. So, as shown in figure 2, we can construct general-purpose computers out of a variety of different systems, including digital circuits and Turing machines (a “thought-experiment” construction of a machine that can follow instructions and read and write symbols on an infinite tape). And systems satisfying the digital circuit abstraction can be constructed from analog circuits, but also from gears or water or light.

Another demonstration of the value of abstracted models is that we can use them, by analogy, to describe quite different systems. So, for example, the constraint models of circuits can be applied to describing transmission of neural signals down the spinal cord, or of heat through a network of connected bodies.

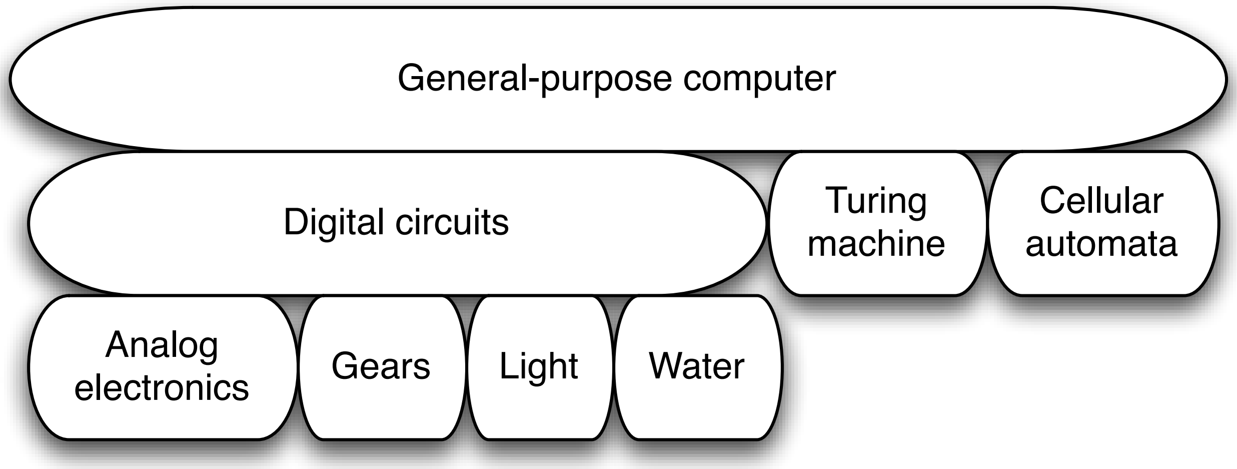


Figure 2: A single abstraction may have a variety of different underlying implementations.

Circuits Typical circuits are built out of a basis set of primative components such as voltage sources, resistors, capacitors, inductors and transistors. This set of component types was chosen to be closely related to primitive concepts in our physical understanding of electronic systems in terms of voltage, current, resistance, and the rate of change of those quantities over time. So, when we buy a physical resistor, we tend to think only of its resistance; and when we buy a capacitor, we think of its ability to store energy. But, of course, that physical resistor has some capacitance, and the capacitor, some resistance. Still, it helps us in our designs to have devices that are as close to the ideal models as possible; and it helps that people have been designing with these components for years, so that we can adopt the strategies the generations of clever people have developed.

The method of combining elements is to connect their terminals together, usually with wire. And our method of abstraction is to describe the *constraints* that a circuit element exerts on the currents and voltages of terminals to which the element is connected.

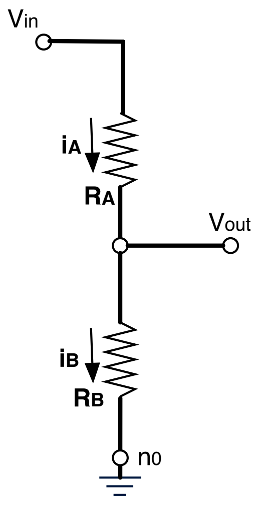
So, armed with the standard basis set of analog circuit components, we can try to build a circuit to control our robot. We have a resistance that varies with the light level, but we need a voltage that does so, as well. We can achieve this by using a *voltage divider*, which is shown in figure 3(a). Using an abstract, constraint-based model of the behavior of circuit components that we'll study in detail later in the course, we can determine the following relationship between V_{out} and V_{in}:

$$V_{\text{out}} = \frac{R_B}{R_A + R_B} V_{\text{in}} \quad .$$

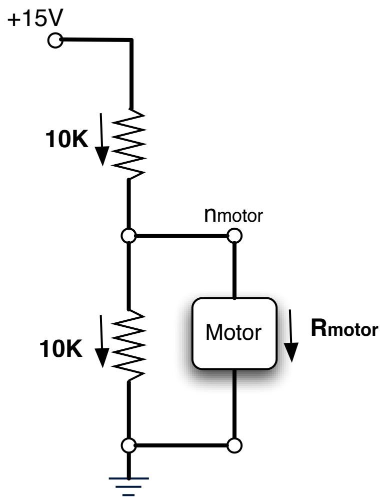
So, for example, if $R_A = R_B$, then $V_{\text{out}} = V_{\text{in}}/2$. Or, in our case, if R_A actually varies with the amount of light shining on it, then V_{out} will also vary with the light.

That's great. So, now, we might imagine that we could use this voltage difference that we've created to drive the motor at different speeds, depending on the light hitting the resistor, using a circuit something like the one shown in figure 3(b). But, sadly, that won't work. It turns out that once we connect the motor to the circuit, it actually changes the voltages.

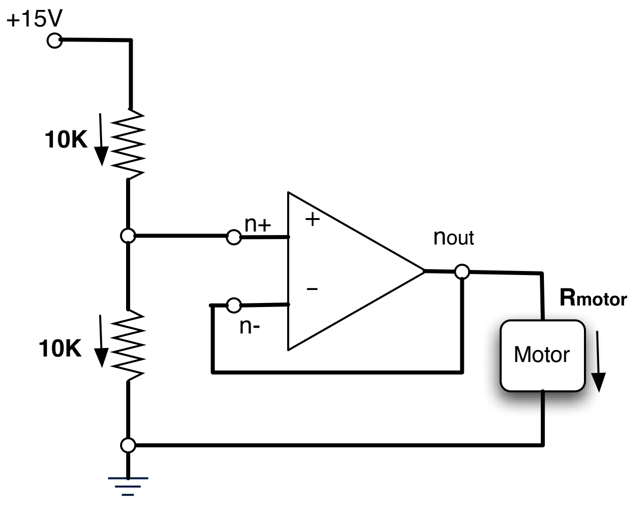
So, although we have developed an abstract model of the behavior of circuit components, which lets us analyze the behavior of a particular complete circuit design, it doesn't give us *modularity.*



(a) Resistor divider.



(b) Connected to a motor.



(c) Connected to a motor with a buffer.

Figure 3: Voltage dividers.

That is, we can't design two parts of a circuit, understand their behavior, and then predict the behavior of the composite system based on the behavior of the separate components. Instead, we'd have to re-analyze the joint behavior of the whole composite system. Lack of modularity makes it very difficult to design large systems, because two different people, or the same person at two different times, cannot design pieces and put them together without understanding the whole.

To solve this problem, we can augment our analog-circuit toolbox with some additional components that allow us to design components with modular behavior; they "buffer" or "isolate" one part of the circuit from another in ways that allow us to combine the parts more easily. In this class, we'll use op-amps to build buffers, which will let us solve our sample problem using a slightly more complex circuit, as shown in figure 3(c).

Digital circuits In analog circuits, we think about voltages on terminals ranging over real values. This gives us the freedom to create an enormous variety of circuits, but sometimes that freedom actually makes the design process more difficult. To design ever more complex circuits, we can move to a much stronger, *digital* abstraction, in which the voltages on the terminals are thought of as only taking on values that are either "low" or "high" (often called 0 and 1). This PCA system is made up of a basis set of elements, called gates, that are built out of simpler analog circuit components, such as transistors and resistors. Adopting the digital abstraction is a huge limitation on the kinds of circuits that can be built, but because it actually simplifies the basic components and increases modularity, it affords designers the cognitive simplicity they need to build incredibly complex machines by designing small parts and putting them together into increasingly larger pieces. Digital watches, calculators, and computers are all built this way.

Elements are combined by wiring, as they were in analog circuits. But in this model, their effects on one another can be isolated to simple input/output relationships at the connections. So now the modularity is very powerful, and each circuit can be described with a Boolean-logic description of how its output values depend on the history of values of its inputs.

Digital design is a very important part of EECS, and it is treated in a number of our courses at basic and advanced levels, but is not one of the topics we'll go into in detail in 6.01.

Computers One of the most important developments in the design of digital circuits is that of the general-purpose "stored program" computer. Computers are a particular class of digital circuits that are general purpose: the same actual circuit can perform (almost) any transformation between its inputs and its outputs. Which particular transformation it performs is governed by a program, which is some settings of physical switches, or information written on an external physical memory, such as cards or a tape, or information stored in some sort of internal memory.

The "almost" in the previous section refers to the actual memory capacity of the computer. Exactly what computations a computer can perform depends on the amount of memory it has; and also on the time you're willing to wait. So, although a general-purpose computer can do anything a special-purpose digital circuit can do, in the information-processing sense, the computer might be slower or use more power. However, using general-purpose computers can save an enormous amount of engineering time and effort. It is *much* cheaper and easier to debug and modify and manufacture software than hardware. The modularities and abstractions that software PCA systems give us are even more powerful than those derived from the digital circuit abstraction.

Python programs Every general-purpose computer has a different detailed design, which means that the way its program needs to be specified is different. Furthermore, the "machine languages" for describing computer programs, at the most primitive level, are awkward for human programmers. So, we have developed a number of computer *programming languages* for specifying a desired computation. These languages are converted into instructions in the computer's native machine language by other computer programs, called compilers or interpreters. One of the coolest and most powerful things about general-purpose computers is this ability to write computer programs that process or create or modify other computer programs.

Computer programming languages are PCA systems. They provide primitive operations, ways of combining them, and ways of abstracting over them. We will spend a considerable amount of time in this course studying the particular primitives, combination mechanisms, and abstraction mechanisms made available in Python. But the choice of Python is relatively unimportant. Virtually all modern programming languages supply a similar set of mechanisms, and the choice of programming language is primarily a matter of taste and convenience.

In a computer program we have both data primitives and computation primitives. At the most basic level, computers generally store binary digits (bits) in groups of 32 or 64, called *words.* These words of data are the primitives, and they can be interpreted by our programs as representing integers, floating-point numbers, strings, or addresses of other data in the computer's memory. The computational primitives supported by most computers include numerical operations such as addition and multiplication, and going and finding the contents of a memory at a given address. In Python, we will work at a level of abstraction that doesn't require us to think very much about addresses of data, but it is useful to understand that level of description, as well.

Primitive data and computation elements can be combined and abstracted in a variety of different ways, depending on choices of programming style. We'll explore these in more detail in section 3.2.

2.3 Models

So far, we have discussed a number of ways of framing the problem of designing and constructing mechanisms. Each PCA system is accompanied by a modeling system, that lets us make mathematical models of the systems we construct.

What is a model? It is a new system that is considerably simpler than the system being modeled, but which captures the important aspects of the original system. We might make a physical model of an airplane to test in a wind tunnel. It doesn't have to have passenger seats or landing lights; but it has to have surfaces of the correct shape, in order to capture the important properties for this purpose. Mathematical models can capture properties of systems that are important to us, and allow us to discover things about the system much more conveniently than by manipulating the original system itself.

One of the hardest things about building models is deciding what aspects of the original system to model and which ones to ignore. Many classic, dramatic engineering failures can be ascribed to failing to model important aspects of the system.

Another important dimension in modeling is whether the model is deterministic or not. We might, for example, model the effect of the robot executing a command to set its velocity as making an instantaneous change to the commanded velocity. But, of course, there is likely to be some delay and some error. We have to decide whether to ignore that delay and error in our model and treat it as if it were ideal; or, for example, to make a probabilistic model of the possible results of that command. Generally speaking, the more different the possible outcomes of a particular action or command, and the more spread out the probability distribution over those outcomes, the more important it is to explicitly include uncertainty in the model.

Once we have a model, it can be used in a variety of different ways, which we explore below.

Analytical models By far the most prevalent use of models is in analysis: Given a circuit design, what will the voltage on this terminal be? Given a computer program, will it compute the desired answer? How long will it take? Given a control system, implemented as a circuit or as a program, will the system stop at the right distance from the light bulb?

Analytical tools are important. It can be hard to verify the correctness of a system by trying it in all possible initial conditions with all possible inputs; sometimes it's easier to develop a mathematical model and then prove a theorem about the model.

For some systems, such as pure software computations, or the addition circuitry in a calculator, it is possible to analyze their correctness or speed with just a model of the system in question. For other systems, such as fuel injectors, it is impossible to analyze the correctness of the controller without also modeling the environment (or "plant") to which it is connected and analyzing the behavior of the coupled system of the controller and environment.

We can do a very simple analysis of a robot moving toward a lamp. Imagine that we arrange it so that the robot's velocity at time t, $V[t]$, is proportional to the difference between the actual light level, $X[t]$ and a desired light level (that is, the light level we expect to encounter when the robot is the desired distance from the lamp), X_{desired}; that is, we can model our control system with the difference equation

$$V[t] = k(X[t] - X_{\text{desired}}) \ .$$

Now, we need to model the world. For simplicity, imagine that the light level is equal (after a units conversion) to the robot's position, and that the robot's position at time t is its position at time $t - 1$ plus its velocity at time $t - 1$. When we couple these models, we get the difference equation

$$X[t] = X[t-1] + k(X[t-1] - X_{\text{desired}}) \ ,$$

where k is the constant of proportionality of the controller. Now, for a given value of k, we can determine how the system will behave over time, by solving the difference equation in X.

Later in the course, we will see how easily determined mathematical properties of a difference equation model can tell us whether a control system will have the desired behavior and whether or not the controller will cause the system to oscillate. These same kinds of analyses can be applied to robot control systems as well as to the temporal properties of voltages in a circuit, and even to problems as unrelated as the consequences of a monetary policy decision in economics.

Synthetic models One goal of many people in a variety of sub-disciplines of computer science and electrical engineering is automatic synthesis of systems from formal descriptions of their desired behavior. For example, you might describe some properties you would want the input/output behavior of a circuit or computer program to have, and then have a computer system discover a circuit or program with the desired property.

This is a well-specified problem, but generally the search space of possible circuits or programs is much too large for an automated process; the intuition and previous experience of an expert human designed cannot yet be matched by computational search methods.

Internal models As we wish to build more and more complex systems with software programs as controllers, we find that it is often useful to build additional layers of abstraction on top of the one provided by a generic programming language. This is particularly true as the nature of the exact computation required depends considerably on information that is only received during the execution of the system.

Consider, for example, an automated taxi robot (or, more prosaically, the navigation system in your new car). It takes, as input, a current location in the city and a goal location, and gives, as output, a path from the current location to the goal. It has a map built into it.

It is theoretically possible to build an enormous digital circuit or computer program that contains a look-up table, so that whenever you get a pair of locations, you can look it up and read out the path. But the size of that table is too huge to contemplate. Instead, what we do is construct an abstraction of the problem as one of finding a path from any start state to any goal state, through a graph (an abstract mathematical construct of "nodes" with "arcs" connecting them). We can develop and implement a general-purpose algorithm that will solve *any* shortest-path problem in a graph. That algorithm and implementation will be useful for a wide variety of possible problems. And for the navigation system, all we have to do is represent the map as a graph, specify the start and goal states, and call the general graph-search algorithm.

We can think of this process as actually putting a model of the system's interactions with the world inside the controller; it can consider different courses of action in the world and pick the one that is the best according to some criterion.

Another example of using internal models is when the system has some significant lack of information about the state of the environment. In such situations, it may be appropriate to explicitly model the set of possible states of the external world and their associated probabilities, and to update this model over time as new evidence is gathered. We will pursue an example of this approach near the end of the course.

3 Programming embedded systems

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. In the following sections, we explore different strategies for building software systems that interact with an external environment, and then different strategies for specifying the purely computational part of a system.

3.1 Interacting with the environment

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.

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 (light sensors, sonars, mouse, keyboard, etc.)
- perform computation, remembering some of the results
- take actions to change the outside world (move the robot, print, draw, etc.)

But these operations can be put together in different styles.

Sequential 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 sonarDistances()[3] > 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.

For the robot moving toward the light, we might write a program like this:

```
while lightValue < desiredValue:
    move(0.1)
```

This would have the root creep up step by step toward the light. We might want to modify it so that the robot moved a distance that was related to the difference between the current and desired light values. However, if it takes larger steps, then during the time that it's moving it won't be sensitive to possible changes in the light value and cannot react immediately to them.

Event-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.

In our simple example, though, we might imagine writing a program that told the robot to drive forward by default; and then install an event handler that says that if the light level reaches the desired value, it should stop. This program would be reactive to sudden changes in the environment.

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.

Transducer An alternative view is that programming a system that interacts with an external world is like building a *transducer* with internal state. Think of a transducer as a processing box that runs continuously. At regular intervals (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:

```
def step():
    distToFront = min(sonarDistances()[3],sonarDistances()[4])
    motorOutput(0.7 * (distToFront - 0.1), 0.0)
```

Executed repeatedly, this program will automatically modulate the robot's speed to be proportional to the free space in front of it.[1] We could make it average over recent sensor values, to smooth out sensor noise, by doing something like:

```
def step():
    distToFront = min(sonarDistances()[3],sonarDistances()[4])
    robot.avgDistToFront = 0.5 * robot.avgDistToFront + 0.5 * distToFront
    motorOutput(0.7 * (avgDistToFront - 0.1), 0.0)
```

We'd have to initialize `robot.avgDistToFront` to some value (maybe 0) before this whole process started.[2]

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). We will start with the transducer model, and then later in the course, we will build a new abstraction layer on top of it that will make it easy to do sequential commands, as well.

3.2 Programming models

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.

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

[1]Here, we are using the commands that our SoaR robot control system uses. The procedure `sonarDistances()` returns a list of distance readings from the sonar sensors; numbers 3 and 4 are looking out the front of the robot. The procedure `motorOutput(fvel, rvel)` tries to set the robot's forward velocity to `fvel` and rotational velocity to `rvel`.

[2]When we want a variable to maintain its value across successive calls to the transducer's step function, we need to prefix it with `robot.` . The reasons for this will become clearer in the next few chapters.

- 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.

Conclusion Python as well as other modern programming languages, such as Java, Ruby and C++, support all of these programming models. The programmer needs to choose which programming model best suits the task. This is an issue that we will return to throughout the course.

Week 1: Python, Functional Programming, Recursion

1 Getting used to 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.

Here are what we think are the most important differences between Python and what you already know about programming.

1.1 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.

1.2 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.[3] If you have a statement that's too long for a line, you can signal it with a backslash:

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

It's easy for Java programmers to get confused about colons and semi-colons in Python. Here's the deal: (1) Python doesn't use semi-colons; (2) Colons are used to start an indented block, so they appear on the first line of a procedure definition, when starting a `while` or `for` loop, and after the condition in an `if`, `elif`, or `else`.

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

1.3 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

[3]Actually, you can write something like `if a > b:  a = a + 1` all on one line, if the work you need to do inside an `if` or a `for` is only one line long.

```
    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.

1.4 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.

1.5 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!**

You should learn enough of Python to be comfortable writing basic programs, and to be able to efficiently look up details of the language that you don't know or have forgotten.

2 Procedures

In Python, the fundamental abstraction of a computation is as a procedure (other books call them "functions" instead; we'll end up using both terms). 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,[4] you might get something like this:

[4] 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.

```
>>> 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

```
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.

Print vs Return Here are two different function definitions:

```
def f1(x):
    print x + 1
def f2(x):
    return x + 1
```

What happens when we call them?

```
>>> f1(3)
4
>>> f2(3)
4
```

It looks like they behave in exactly the same way. But they don't, really. Look at this example:

```
>>> print(f1(3))
4
None
>>> print(f2(3))
4
```

In the case of `f1`, the function, when evaluated, prints 4; then it returns the value `None`, which is printed by the Python shell. In the case of `f2`, it doesn't print anything, but it returns 4, which is printed by the Python shell. Finally, we can see the difference here:

```
>>> f1(3) + 1
4
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>> f2(3) + 1
5
```

In the first case, the function doesn't return a value, so there's nothing to add to 1, and an error is generated. In the second case, the function returns the value 4, which is added to 1, and the result, 5, is printed by the Python read-eval-print loop.

The book *How to Think Like a Computer Scientist*, which we recommend reading, is translated from a version for Java, and it has a lot of `print` statements in it, to illustrate programming concepts. But for just about everything we do, it will be returned values that matter, and printing will be used only for debugging, or to give information to the user.

3 Execution model

You all have an informal understanding of what happens when your programs are executed. Usually, the informal understanding is enough to let you write effective programs. But it's useful to really understand what's going on, both because the principles are interesting and important and because it will help you predict what your program will do if it's more complicated or differently structured than usual.

We'll explore exactly what happens when a Python program is executed, in the sections below.

3.1 Environments

The first thing we have to understand is the idea of *binding environments* (we'll often just call them *environments*; they are also called *namespaces* and *scopes*). An environment is a stored mapping between names and entities in a program. The entities can be all kinds of things: numbers, strings, lists, procedures, objects, etc. In Python, the names are strings and environments are actually dictionaries, which we've already experimented with.

Environments are used to determine values associated with the names in your program. There are two operations you can do to an environment: add a binding, and look up a name. You do these things implcitly all the time in programs you write. Consider a file containing

```
a = 5
print a
```

The first statement, `a = 5`, creates a binding of the name `a` to the value 5. The second statement prints something. First, to decide that it needs to print, it looks up `print` and finds an associated built-in procedure. Then, to decide what to print, it evaluates the associated expression. In this case, the expression is a name, and it is evaluated by looking up the name in the environment and returning the value it is bound to (or generating an error if the name is not bound).

In Python, there are environments associated with each module (file) and one called `__builtin__` that has all the procedures that are built into Python. If you do

```
>>> import __builtin__
>>> dir(__builtin__)
```

you'll see a long list of names of things (like `'sum'`), which are built into Python, and whose names are defined in the builtin module. You don't have to type `import __builtin__`; as we'll see below, you always get access to those bindings. You can try importing `math` and looking to see what names are bound there.

Another operation that creates a new environment is a function call. In this example,

```
def f(x):
    print x
>>> f(7)
```

when the function `f` is called with argument 7, a new *local* environment is constructed, in which the name `x` is bound to the value 7.

So, what happens when Python actually tries to evaluate `print x`? It takes the symbol `x` and has to try to figure out what it means. It starts by looking in the *local* environment, which is the one defined by the innermost function call. So, in the case above, it would look it up and find the value 7 and return it.

Now, consider this case:

```
def f(a):
    def g(x):
        print x, a
        return x + a
    return g(7)
>>> f(6)
```

What happens when it's time to evaluate `print x, a`? First, we have to think of the environments. The first call, `f(6)` establishes an environment in which `a` is bound to 6. Then the call `g(7)` establishes another environment in which `x` is bound to 7. So, when needs to print `x` it looks in the local environment and finds that it has value `7`. Now, it looks for `a`, but doesn't find it in the local environment. So, it looks to see if it has it available in an *enclosing environment*; an environment that was enclosing this procedure *when it was defined.* In this case, the environment associated with the call to `f` is enclosing, and it has a binding for `a`, so it prints `6` for `a`. So, what does `f(6)` return? 13.

You can think of every environment actually consisting of two things: (1) a dictionary that maps names to values and (2) an enclosing environment.

If you write a file that looks like this:

```
b = 3
def f(a):
    def g(x):
        print x, a, b
        return x + a + b
    return g(7)

>>> f(6)
7 6 3
16
```

When you evaluate `b`, it won't be able to find it in the local environment, or in an enclosing environment created by another procedure definition. So, it will look in the *global environment*. The name global is a little bit misleading; it means the environment associated with the file. So, it will find a binding for `b` there, and use the value 2.

One way to remember how Python looks up names is the LEGB rule: it looks, in order, in the *Local*, then the *Enclosing*, then the *Global*, then the *Builtin* environments, to find a value for a name. As soon as it succeeds in finding a binding, it returns that one.

We have seen two operations that cause bindings to be created: assignments, and function calls. While we're talking about assignments, here's a nice trick, based on the packing and unpacking of *tuples*. First, we have to teach you about tuples, which are like lists, but whose contents can't be changed once they are created. To make a tuple, you use the comma operator:

```
>>> x = 1, 2, 3
>>> x
(1, 2, 3)
>>> x[1]
2
```

One slightly weird-looking thing is that the way to make a tuple with a single element is to use a single comma (not to put it inside round parentheses; parentheses are just used for grouping not for constructing tuples). So,

```
>>> y = 1,
>>> y
(1,)
>>> len(y)
1
```

Now, here's the cool assignment trick:

```
>>> a, b, c = 1, 2, 3
>>> a
1
>>> b
2
>>> c
3
```

Or, with lists,

```
>>> [a, b, c] = [1, 2, 3]
>>> a
1
>>> b
2
>>> c
3
```

When you have a list (or a tuple) on the left-hand side of an assignment statement, you have to have a list (or tuple) of matching structure on the right-hand side. Then Python will "unpack" them both, and assign to the individual components of the structure on the left hand side. You can get fancier with this method:

```
thing = [8, 9, [1, 2], 'John', [33.3, 44.4]]
>>> [a, b, c, d, [e1, e2]] = thing
>>> c
[1, 2]
>>> e1
33.299999999999997
```

Bindings are also created when you execute an `import` statement. If you execute

```
import math
```

Then the `math` module is loaded and the name `math` is bound, in the current environment, to the math module. No other names are added to the current environment, and if you want to refer to internal names in that module, you have to qualify them, as in `math.sqrt`. If you execute

```
from math import sqrt
```

then, the `math` module is loaded, and the name `sqrt` is bound, in the current environment, to whatever the the name `sqrt` is bound to in the `math` module. But note that if you do this, the name `math` isn't bound to anything, and you can't access any other procedures in the `math` module.

Another thing that creates a binding is the definition of a function: that creates a binding of the function's name, in the environment in which it was created, to the actual function.

Finally, bindings are created by `for` statements and list comprehensions; so, for example,

```
a = 1
b = 2
c = 3
def d(a):
    c = 5
    from math import pi
    def e(x):
        for i in range(b):
            print a, b, c, x, i, pi, d, e
    e(100)

>>> d(1000)
1000 2 5 100 0 3.14159265359 <function d at 0x4e3f0> <function e at 0x4ecf0>
1000 2 5 100 1 3.14159265359 <function d at 0x4e3f0> <function e at 0x4ecf0>
```

Figure 4: Code and what it prints

```
for element in listOfThings:
    print element
```

creates successive bindings for the name `element` to the elements in `listOfThings`.

Figure 5 shows the state of the environments when the print statement in the example code shown in figure 4 is executed. Names are first looked for in the local scope, then its enclosing scope (if there were more than one enclosing scope, it would continue looking up the chain of enclosing scopes), and then in the global scope.

Local versus global references There is an important subtlety in the way names are handled in handled in the environment created by a procedure call. When a name that is not bound in the local environment is referred to, then it is looked up in the enclosing, global, and built-in environments. So, as we've seen, it is fine to have

```
a = 2
def b():
    print a
```

When a name is assigned in a local environment, a new binding is created for it. So, it is fine to have

```
a = 2
def b():
    a = 3
    c = 4
    print a, c
```

Both assignments cause new bindings to be made in the local environment, and it is those bindings that are used to supply values in the print statement.

But here is a code fragment that causes trouble:

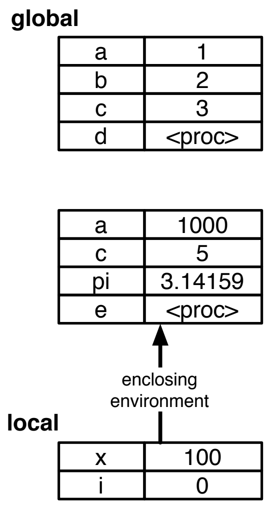


Figure 5: Binding environments that were in place when the first print statement in figure 4 was executed.

```
a = 3
def b():
    a = a + 1
    print a
```

It seems completely reasonable, and you might expect it to print 4. But, instead, it generates an error if we try to call `b`. What's going on?? It all has to do with when Python decides to add a binding to the local environment. When it sees this procedure definition, it sees that the name `a` is assigned to, and so, at the very beginning, it puts an entry for `a` in the local environment. Now, when it's time to execute the statement

```
a = a + 1
```

it starts by evaluating the expression on the right hand side: `a + 1`. When it tries to look up the name `a` in the local environment, it finds that it has been added to the environment, but hasn't yet had a value specified. So it generates an error.

We can still write code to do what we intended (write a procedure that increments a number named in the global environment), by using the `global` declaration:

```
a = 3
def b():
    global a
    a = a + 1
    print a
>>> b()
4
>>> b()
5
```

The statement `global a` asks that a new binding for `a` *not* be made in the local environment. Now, all references to `a` are to the binding in the global environment, and it works as expected. In Python, we can only make assignments to names in the local scope or in the global scope, but not to names in an enclosing scope. So, for example,

```
def outer():
    def inner():
        a = a + 1
    a = 0
    inner()
```

In this example, we get an error, because Python has made a new binding for `a` in the environment for the call to `inner`. We'd really like for `inner` to be able to see and modify the `a` that belongs to the environment for `outer`, but there's no way to arrange this.

3.2 Evaluating expressions

Now, we're ready to understand what Python does when it evaluates an expression, such as `f(3, g(a, 5))`. The evaluator takes an expression and an environment and returns a value; the value can be a number, string, dictionary, list, procedure, etc. Evaluation in Python can be nicely described as a recursive process.

There are two base cases:

1. If the expression is a constant (number or string), then the value is just that constant value;
2. If the expression is a name, then the value is the result of looking the name up in the local, enclosing, global, and built-in environments until the first binding is found, and the associated value is returned.

In the recursive case, we have the application of a function to some arguments.

- We start by evaluating the expressions that describe the function and each of the arguments (by making a recursive call to the evaluator). At this point, we have our hands on an actual function and a set of values that should be passed in as values to the function.
- Next, we establish a new environment, in which the formal parameters (the argument names in the definition) of the function are bound to the values we just computed.
- Now, we evaluate the body of the procedure in that new environment (by making a recursive call to the evaluator)

This is a beautiful process that is actually quite easy to implement for simple functional languages.

In Python, our expressions often look more like

```
3 + (a*4)
```

The infix operators `+` and `*` are just special syntax, and that expression gets converted internally by Python into something like

```
operator.plus(3,operator.times(a,4))
```

before it is evaluated in the way described above.

3.3 Special forms

Of course, Python is more than expression evaluation and assignment. In order to have a truly general-purpose programming language, we need at least one more construct: `if`. The important thing about `if` is that it either evaluates one set of statements or another depending on whether the condition is true. You can't make that happen simply by evaluating functions using the model above.

Python supplies all sorts of other handy things, like loops, and list comprehensions. We could precisely specify the execution model for each of these statements (and such a language specification exists for Python), and then it's a medium-sized but well-defined job to write an interpreter for Python that does what the shell does: reads in Python programs and does what they specify.

4 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.

4.1 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.[5]

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
```

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
```

[5]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.

will print `0 1 2`. 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
```

Later, 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.

5 Functional Style

In the functional programming style, one tries to use a very small set of primitives and means of combination. We'll see that recursion is a very powerful primitive, which could allow us to dispense with all other looping constructs (`while` and `for`) and results in code with a certain beauty.

Another element of functional programming style is the idea of functions as *first-class objects.* That means that we can treat functions or procedures in much the same way we treat numbers or strings in our programs: we can pass them as arguments to other procedures and return them as the results from procedures. This will let us capture important common patterns of abstraction and will also be an important element of object-oriented programming.

5.1 Basics

But let's begin at the beginning. The primitive elements in the functional programming style are basic functions. They're combined via function composition: so, for instance `f(x, g(x, 2))` is a composition of functions. To abstract away from the details of an implementation, we can define a function

```
def square(x):
    return x*x
```

Or

```
def average(a,b):
    return (a+b)/2.0
```

And now, having defined those functions, we can use them exactly as if they were primitives. So, for example, we could compute the mean square of two values as

```
average(square(7),square(11))
```

We can also construct functions "anonymously" using the `lambda` constructor:

```
>>> f = lambda x: x*x
>>> f
<function <lambda> at 0x4ecf0>
>>> f(4)
16
```

The word lambda followed by a sequence of variables separated by commas, then a colon, then a single expression using those variables, defines a function. It doesn't need to have an explicit `return`; the value of the expression is always returned. A single expression can only be one line of Python, such as you could put on the right hand side of an assignment statement. Here are some other examples of functions defined using `lambda`.

This one uses two arguments.

```
>>> g = lambda x,y : x * y
>>> g(3, 4)
12
```

You don't need to name a function to use it. Here we construct a function and apply it all in one line:

```
>>> (lambda x,y : x * y) (3, 4)
12
```

Here's a more interesting example. What if you wanted to compute the square root of a number? Here's a procedure that will do it:

```
def sqrt(x):
    def goodEnough(guess):
        return abs(x-square(guess)) < .0001
    guess = 1.0
    while not goodEnough(guess):
        guess=average(guess,x/guess)
    return guess

>>>sqrt(2)
1.4142156862745097
```

This is an ancient algorithm, due to Hero of Alexandria.[6] It is an *iterative* algorithm: it starts with a guess about the square root, and repeatedly asks whether the guess is good enough. It's good enough if, when we square the guess, we are close to the number, `x`, that we're trying to take the square root of. If the guess isn't good enough, we need to try to improve it. We do that by making a new guess, which is the average of the original guess and `x / guess`.

How do we know that this procedure is ever going to finish? We have to convince ourself that, eventually, the guess will be good enough. The mathematical details of how to make such an argument are more than we want to get into here, but you should at least convince yourself that the new guess is closer to `x` than the previous one was. This style of computation is generally quite useful. We'll see later in this chapter how to capture this *common pattern* of function definition and re-use it easily.

Note also that we've defined a function, `goodEnough`, inside the definition of another function. We defined it internally because it's a function that we don't expect to be using elsewhere; but we think naming it makes our program for `sqrt` easier to understand. We'll study this style of definition in detail in the next chapter. For now, it's important to notice that the variable `x` inside `goodEnough` is the same `x` that was passed into the `sqrt` function.

5.2 List Comprehensions

Python has a very nice built-in facility for doing many interative operations called *list comprehensions.* The general template is
`[`*expr* `for` *var* `in` *list*`]`
where *var* is a single variable, *list* is an expression that results in a list, and *expr* is an expression that uses the variable *var*. The result is a list, of the same length as *list*, which is obtained by successively letting *var* take values from *list*, and then evaluating *expr*, and collecting the results into a list.

Whew. It's probably easier to understand it by example.

```
>>> [x/2.0 for x in [4, 5, 6]]
[2.0, 2.5, 3.0]
>>> [y**2 + 3 for y in [1, 10, 1000]]
[4, 103, 1000003]
```

[6]Hero of Alexandria is involved in this course in multiple ways. We'll be studying feedback later on in the course, and Hero was the inventor of several devices that use feedback, including a self-filling wine goblet. Maybe we'll assign that as a final project...

```
>>> [a[0] for a in [['Hal', 'Abelson'],['Jacob','White'],
                    ['Leslie','Kaelbling']]]
['Hal', 'Jacob', 'Leslie']
>>> [a[0]+'!' for a in [['Hal', 'Abelson'],['Jacob','White'],
                        ['Leslie','Kaelbling']]]
['Hal!', 'Jacob!', 'Leslie!']
>>> [x/2.0 for x in [4, 5, 6]]
[2.0, 2.5, 3.0]
```

Another useful feature of list comprehensions is that they implement a common functional-programming capability of *filtering*. Imagine that you have a list of numbers and you want to construct a list containing just the ones that are odd. You might write

```
>>> nums = [1, 2, 5, 6, 88, 99, 101, 10000, 100, 37, 101]
>>> [x for x in nums if x%2==1]
[1, 5, 99, 101, 37, 101]
```

And, of course, you can combine this with the other abilities of list comprehensions, to, for example, return the squares of the odd numbers:

```
>>> [x*x for x in nums if x%2==1]
[1, 25, 9801, 10201, 1369, 10201]
```

6 Recursion

In the previous example, we defined a `sqrt` function, and now we can use it without remembering or caring about the details of how it is implemented. We sometimes say that we can treat it as a *black box*, meaning that it is unnecessary to look inside it to use it. This is crucial for maintaining sanity when building complex pieces of software. An even more interesting case is when we can think of the procedure that we're in the middle of defining as a black box. That's what we do when we write a recursive procedure.

Recursive procedures are ways of doing a lot of work. The amount of work to be done is controlled by one or more arguments to the procedure. The way we are going to do a lot of work is by calling the procedure, over and over again, from inside itself! The way we make sure this process actually terminates is by being sure that the argument that controls how much work we do gets smaller every time we call the procedure again. The argument might be a number that counts down to zero, or a string or list that gets shorter.

There are two parts to writing a recursive procedure: the base case(s) and the recursive case. The *base case* happens when the thing that's controlling how much work you do has gotten to its smallest value; usually this is 0 or the empty string or list, but it can be anything, as long as you know it's sure to happen. In the base case, you just compute the answer directly (no more calls to the recursive function!) and return it. Otherwise, you're in the *recursive* case. In the recursive case, you try to be as lazy as possible, and foist most of the work off on another call to this function, but with one of its arguments getting smaller. Then, when you get the answer back from the recursive call, you do some additional work and return the result.

Here's an example recursive procedure that returns a string of n 1's:

```
def bunchaOnes(n):
    if n == 0:
        return "
    else:
        return bunchaOnes(n-1) + ‘‘1’’
```

The thing that's getting smaller is `n`. In the base case, we just return the empty string. In the recursive case, we get someone else to figure out the answer to the question of `n-1` ones, and then we just do a little additional work (adding one more `‘‘1’’` to the end of the string) and return it.

Here's another example. It's kind of a crazy way to do multiplication, but logicians love it.

```
def mult(a,b):
    if a==0:
        return 0
    else:
        return b + mult(a-1,b)
```

Trace through an example of what happens when you call `mult(3, 1)`, by adding a print statement as the first line of the function that prints out its arguments, and seeing what happens.

Here's a more interesting example of recursion. Imagine we wanted to compute the binary representation of an integer. For example, the binary representation of 145 is '10010001'. Our procedure will take an integer as input, and return a string of 1's and 0's.

```
def bin(n):
    if n == 0:
        return '0'
    elif n == 1:
        return '1'
    else:
        return bin(n/2) + bin(n%2)
```

The easy cases (base cases) are when we're down to a 1 or a 0, in which case the answer is obvious. If we don't have an easy case, we divide up our problem into two that are easier. So, if we convert `n/2` into a string of digits, we'll have all but the last digit. And `n%2` is 1 or 0 depending on whether the number is even or odd, so one more call of `bin` will return a string of '0' or '1'. The other thing that's important to remember is that the `+` operation here is being used for string concatenation, not addition of numbers.

How do we know that this procedure is going to terminate? We know that the number it's operating on is getting smaller and smaller, and will eventually be either a 1 or a 0, which can be handled by the base case.

You can also do recursion on lists. Here's another way to do our old favorite `addList`:

```
def addList6(list):
    if list == []:
        return 0
    else:
        return list[0] + addList6(list[1:])
```

There's a new bit of syntax here: `list[1:]` gives all but the first element of `list`. Go read the section in the Python documentation on subscripts to see how to do more things with list subscripts.

The `addList` procedure consumed a list and produced a number. The `incrementElements1` procedure below shows how to use recursion to do something to every element of a list and make a new list containing the results.

```
def incrementElements1(elts):
    if elts == []:
        return []
    else:
        return [elts[0]+1] + incrementElements1(elts[1:])
```

7 Shared structure

When we work with lists, it is important to know when we are working with multiple references to the same list. You can think of a list as a container with some elements in it. It is possible to have two different containers with exactly the same elements. Let's explore these ideas a bit.

In the listing below, `a` and `b` are names that both refer to the same list. So, if we change `a`, we change `b`.

```
a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[1] = 100
>>> a
[1, 100, 3]
>>> b
[1, 100, 3]
```

Now, let's make a new list with the same elements, and see what happens when we make various comparisons:

```
>>> c = [1, 100, 3]
>>> a == b
True
>>> a == c
True
>>> a is b
True
>>> a is c
False
```

There are really two different equality tests in Python. The `==` test, when applied to lists, tests to see if the contents of the container are the same. So `a == c` is true. But `is` tests to see if the actual containers are the same. In our case, `a` is the same as `b`, but not the same as `c`.

It's important to keep track of which operations on lists actually change the existing list and which make a copy of it. We can see from the following transcript that when we use * to append several copies of a list, it makes all new lists structures:

```
>>> d = 2*a
>>> d
[1, 100, 3, 1, 100, 3]
>>> a[1] = 9
>>> a
[1, 9, 3]
>>> d
[1, 100, 3, 1, 100, 3]
```

But if we make a list containing a, twice, the list has the same container in it twice, and it doesn't copy the list structure.

```
>>> e = [a, a]
>>> e
[[1, 9, 3], [1, 9, 3]]
>>> a[1] = 200
>>> e
[[1, 200, 3], [1, 200, 3]]
```

Note that 2*[a] is the same as [a,a].

You can experiment with other list-manipulation procedures in Python, to see whether they make copies of lists or just refer to the containers. You will find that append, sort and reverse actually change the structure of the lists they operate on. However, slicing (getting a sub-list) gives you a whole new list structure. So, changing one of y or x in the example below won't change the other one.

```
>>> x = [1, 2, 3]
>>> y = x[1:]
>>> y
[2, 3]
>>> x[2] = 100
>>> x
[1, 2, 100]
>>> y
[2, 3]
```

This leads us to an important idea: if you have a list named z and you want to make a copy of it (because, for example, you're about to change it, but you want the original around, too), then you can just do z[:], and you'll get a completely new container with the same elements in it.

```
z = [1, 2, 3, 4]
>>> zcopy = z[:]
>>> zcopy
```

```
[1, 2, 3, 4]
>>> z[0] = 'ha'
>>> z
['ha', 2, 3, 4]
>>> zcopy
[1, 2, 3, 4]
```

One more thing to know about and watch out for is procedures that actually change their arguments. In *pure* functional programming, procedures never change anything; they always return a newly made structure or part of an old structure. In a pure functional program, in fact, it doesn't matter whether your structures are shared. Any computation that can be done by a computer can be done by a pure functional program with no assignment statements or changes to structures, but it can be *wildly* inefficient, in many cases, to do so.

So, we may end up writing procedures that change the values of parameters passed into them. In fact, you cannot change the entire value of a parameter; so, for instance, this `changer` procedure doesn't actually change anything (as we'll see next week, it makes a new internal variable called `x`).

```
>>> a = 6
>>> def changer(x):
...     x = 4
>>> a
6
>>> changer(a)
>>> a
6
```

However, we can change values contained in a list structure that is passed in as a parameter:

```
>>> def changeFirstEntry(x):
...     x[0] = 'nya nya'
>>> b = [1, 2, 3]
>>> changeFirstEntry(b)
>>> b
['nya nya', 2, 3]
```

There is nothing wrong with doing this; in fact it's often a good way to structure programs. But whenever you write a procedure that changes the contents of one of its arguments, you should take care to document that fact as obviously as possible.