The Digital World

Singapore University of Technology & Design

January 2013

Contents

Chapter 1

Introduction

1.1 Goals

We have many goals for this course. Our primary goal is for you to learn to appreciate and use the fundamental design principles of modularity and abstraction in a variety of contexts. To achieve this goal, we will learn how to build 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 to show you that making mathematical models of real systems can help in the design and analysis of those systems; and to give you practice with the difficult step of deciding which aspects of the real world are important to the problem being solved and how to model them 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. 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 an 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.

Finally, of course, we have the more typical goals of teaching exciting and important basic material, including modern software engineering, linear systems analysis, and decision-making. This material has an internal elegance and beauty, and plays a crucial role in building many modern systems.

1.2 Modularity, Abstraction, and Modeling

Whether proving a theorem by building up from lemmas to basic theorems to more specialized results, 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. Without such tools, a single person would be overwhelmed by the complexity of a system, as there is only so much detail that a single person can consciously manage 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 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 changes the direction of motion).

Given basic modules, 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. And, of course, this process can be repeated over many stages. 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.

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 and control systems.

1.2.1 An Abstraction Hierarchy of Mechanisms

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 is important, as a designer, to develop a repertoire of standard bits and 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 valuable techniques from them, rather than having to re-invent the techniques 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;
- ways of "packaging" or *abstracting* pieces of a design so they can be reused (in essence creating new "primitives"); and

- ways of capturing common *patterns* of abstraction (essentially, abstracting our abstractions).

Very complicated design problems can become tractable using such a *primitive-combination-abstraction-pattern* (PCAP) approach. In this class, we will examine and learn to use a variety of PCAP strategies. In the rest of this section, we will hint at some of the PCAP systems we will be developing in much greater depth throughout the class. Figure 1.1 shows one view of this development, as a successive set of restrictions of the design space of mechanisms.

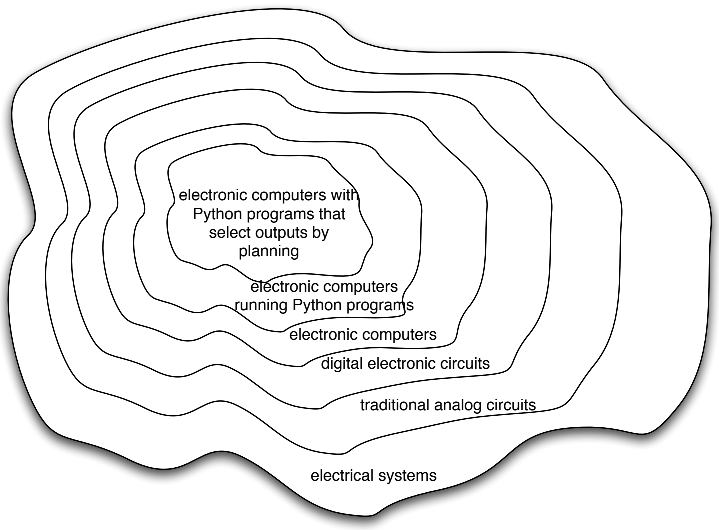

electronic computers with Python programs that select outputs by planning
electronic computers running Python programs
electronic computers
digital electronic circuits
traditional analog circuits
electrical systems

Figure 1.1: Increasingly constrained systems.

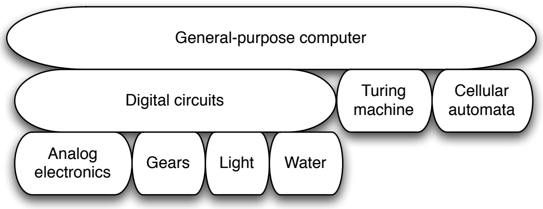


Figure 1.2: A single abstraction may have a variety of different underlying implementations.

One very important thing about abstract models is that once we have fixed the abstraction, it will usually be possible to implement it using a variety of different underlying substrates. So, as shown in Figure 1.2, we can construct general-purpose computers out of a variety of different kinds of systems, including digital circuits and general-purpose computers. And systems satisfying the digital circuit abstraction can be constructed from analog circuits, but also from gears 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 that we will study can be applied to describing transmission of neural signals down the spinal cord, or of heat through a network of connected bodies.

Let's explore part of the abstraction hierarchy in Figure 1.1, from computers to Python programs, in more detail, moving up abstraction levels while observing common patterns.

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 are 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 PCAP systems give us are even more powerful than those derived from the digital circuit abstraction.

Again, we can see how abstraction separates use from details; we don't need to know how the circuits inside a computer are designed, we just need to know the rules by which we can use them and the constraints under which they perform.

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 PCAP systems. They provide primitive operations, ways of combining them, ways of abstracting over them, and ways of capturing common abstraction patterns. We will spend a considerable amount of time in this course studying the particular primitives, combination mechanisms, and abstraction mechanisms made available in the Python programming language. 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 are data 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 locating and extracting the contents of a memory at a given address. In Python, we will work at a level of abstraction that does not 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 will explore these in more detail in Section 1.3.2.

1.2.2 Models

So far, we have discussed a number of ways of framing the problem of designing and constructing mechanisms. Each PCAP 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 does not 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 which 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. One example (which turned out to be an ethical success story, rather than a failure) is LeMessurier's Citicorp Building[1], in which an engineering design change was made, but tested in a model in which the wind came only from a limited set of directions.

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 diffuse the probability distribution over those outcomes, the more important it is to explicitly include uncertainty in the model.

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

[1]See "The Fifty-Nine Story Crisis", *The New Yorker*, May 29, 1995, pp 45-53.

Analytical Models

By far the most prevalent use of models is in analysis: Given a computer program, will it compute the desired answer? How long will it take? Given a control system, implemented as a program, will the system stop at the right distance from a wall?

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 is easier to develop a mathematical model and then prove a theorem about the model.

For some systems, such as pure software computations, it is possible to analyze 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 then analyzing the behavior of the coupled system of the controller and environment.

To demonstrate some of these tradeoffs, 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

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

where k is the constant of proportionality, or *gain*, of the controller.

Now, we need to model the world. For simplicity, we equate the light level to the robot's position (assuming that the units match); in addition we assume the robot's position at time t is its position at time $t-1$ plus its velocity at time $t-1$ (actually, this should be the product of the velocity and the length of time between samples, but we can just assume a unit time step and thus use velocity). When we couple these models, we get the difference equation

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

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 the controller will cause the system to oscillate. These same kinds of analyses can be applied to robot control systems, and even to problems as unrelated as the consequences of a monetary policy decision in economics. It is important to note that treating the system as moving in discrete time steps is an approximation to underlying continuous dynamics; it can make models that are easier to analyze, but it requires sophisticated understanding of sampling to understand the effects of this approximation on the correctness of the results.

Synthetic Models

One goal of many engineers is the 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 computer program to have, and then have a computer system discover a 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 designer cannot yet be matched by computational search methods.

However, humans use informal models of systems for synthesis. The documentation of a software library, which describes the function of the various procedures, serves as an informal model that a programmer can use to assemble those components to build new, complex systems.

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 when 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 computer program that contains a look-up table, in which we precompute the path for all pairs of locations and store them away. Then when we want to find a path, we could simply look in the table at the location corresponding to the start and goal location and retrieve the precomputed path. Unfortunately, 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.

1.3 Programming Embedded Systems

There are many 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.

1.3.1 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, so that as the system is computing, the world is changing, requiring future computation to adapt to the new state of the world.

There are a variety of different ways to organize 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, and
- take actions to change the outside world (move the robot, print, draw, etc.).

These operations can be put together in different styles.

1.3.1.1 Sequential

The most immediately straightforward style for constructing a program that interacts with the world is the basic imperative style, in which the program gives a sequence of 'commands' to the system it is controlling. 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 noObstacleInFront:
    moveDistance(1)
    turnAngle(90)
    moveDistance(1)
    turnAngle(90)
    moveDistance(1)
    turnAngle(90)
    moveDistance(1)
    turnAngle(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 the robot, either because someone walked in front of it 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:
    moveDistance(0.1)
```

This would have the robot 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 is moving it will not be sensitive to possible changes in the light value and cannot react immediately to them.

1.3.1.2 Event-Driven

User-interface programs are often best organized differently, as *event-driven* (also called *interrupt driven*) programs. In this case, the program is specified as a collection of procedures (called 'handlers' or 'callbacks') that are attached to particular events that can take place. So, for example, there might be procedures 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 whether any of the triggering events have happened, and, if they have, calling the associated procedure.

In our simple example, 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 both keep a program like this running well and guarantee a minimum response time to any event.

1.3.1.3 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 (perhaps 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 defining a procedure:

```
def step():
    distToFront = min(frontSonarReadings)
    motorOutput(gain * (distToFront - desiredDistance), 0.0)
```

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

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, as described in Section 4.3, we will build a new abstraction layer on top of it that will make it easy to do sequential commands, as well.

1.3.2 Programming Models

Just as there are different strategies for organizing entire software systems, there are different strategies for formally expressing computational processes within those structures.

1.3.2.1 Imperative Computation

Some 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.

1.3.2.2 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 would not 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), the `if` operator, 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.

1.3.2.3 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 work directly with all of these primitives, except for addresses.

There is only so much you can do with a single number, though. We would 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.

1.3.2.4 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 systems 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.

Programming Languages

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.

1.4 Summary

We hope that this course will give you a rich set of conceptual tools and practical techniques, as well as an appreciation of how math, modeling, and implementation can work together to enable the design and analysis of complex computational systems that interact with the world.

Chapter 2

Learning to Program in Python

Depending on your previous programming background, we recommend different paths through Chapters 2 and 3:

- **If you have never programmed before:** you should start with a general introduction to programming and Python. We recommend *Python for Software Design: How to Think Like a Computer Scientist*, by Allen Downey. This is a good introductory text that uses Python to present basic ideas of computer science and programming. It is available for purchase in hardcopy, or as a free download from:

  ```
  http://www.greenteapress.com/thinkpython
  ```

  After that, you can go straight to the next chapter. In the Digital World course, we assume you have **no programming experience**, and will spend the first six weeks teaching you how to program in Python.
- **If you have programmed before, but not in Python:** you should read the rest of this chapter for a quick overview of Python, and how it may differ from other programming languages with which you are familiar.
- **If you have programmed in Python:** you can skip to the next chapter, but we still recommend that you skim through this chapter.

Everyone should have a bookmark in their browser for *Python Tutorial*, by Guido Van Rossum. This is the standard tutorial reference by the inventor of Python. It is accessible at:

```
http://docs.python.org/tut/tut.html
```

In the rest of this chapter, we will assume you know how to program in some language, but are new to Python. We will use Java as an informal running comparative example. In this section we will cover what we think are the most important differences between Python and what you may already know about programming; but these notes are by no means complete.

2.1 Using Python

Python is designed for easy interaction between a user and the computer. 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 entered, 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.

2.1.1 Indentation and Line Breaks

Every programming language has to have some method for indicating grouping of instructions. Here is 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 is not required. In addition, the semi-colons are used to indicate the end of a statement, independent of the locations of the line breaks in the file. So, the following code fragment has the same meaning as the previous one, although it is much harder to read and understand.

```
    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 would write the previous example as

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

If you have a statement that is too long for a line, you can signal it with a backslash:

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

It is easy for Java programmers to get confused about colons and semi-colons in Python. Here is the deal: (1) Python does not 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 is entirely a matter of taste. But if you are going to use Python, you need to remember that indentation and line breaks are significant.

2.1.2 Types and Declarations

Java programs are what is known as *statically and strongly typed*. Thus, 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 are used. It also means that the variables cannot be used in a way that is inconsistent with their type. So, for instance, you would declare `x` to be an integer by saying

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

But you would 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, the following 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 certain about what kind of data is stored in each variable, and it does not 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 could not evaluate

```
3 + "hi"
```

Python would not 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 is easy to make a list or array with objects of different types stored in it. In Java, it can be done, but it is 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.

2.1.3 Modules

As you start to write bigger programs, you will want to keep the procedure definitions in multiple files, grouped together according to what they do. So, for example, we 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 we want to use those procedures in another file, or from the the Python shell, we will need to say

```
import utility
```

so that all those procedures become available to us and to the Python interpereter. Now, if we have a procedure in `utility.py` called `foo`, we can use it with the name `utility.foo`. You can read more about modules, and how to reference procedures defined in modules, in the Python documentation.

2.1.4 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 is much easier to test the individual pieces as you go, rather than to spend hours writing a big program, and then find it does not 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, do not despair. Debugging a program does not 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 are 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. Do not change any code until you know what is wrong with what you are doing now, and therefore believe that the change you make is going to correct the problem.

Ultimately, for debugging big programs, it is 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 will learn to debug systematically using "print" statements.

One good way to use "print" statements to help in debugging is to use a variation on binary search. Find a spot roughly halfway through your code at which you can predict the values of variables, or intermediate results of your computation. Put a print statement there that lists expected as well as actual values of the variables. Run your test case, and check. If the predicted values match the actual ones, it is likely that the bug occurs after this point in the code; if they do not, then you have a bug prior to this point (of course, you might have a second bug after this point, but you can find that later). Now repeat the process by finding a location halfway between the beginning of the procedure and this point, placing a print statement with expected and actual values, and continuing. In this way you can narrow down the location of the bug. Study that part of the code and see if you can see what is wrong. If not, add some more print statements near the problematic part, 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.

2.2 Procedures

In Python, the fundamental abstraction of a computation is a procedure (other books call them “functions” instead; we will 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,[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 is 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` did not return anything. Well...strictly speaking, it returned a special value called `None`, which the shell does not bother printing out. The value `None` has a

[1]Although you can type procedure definitions directly into the shell, you will not want to work that way, because if there is a mistake in your definition, you will 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 Idle for an explanation of how to do that.

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 did not know how to add something of type `NoneType` and something of type `int`.

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

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 does not 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 does not return a value, so there is 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 *Think Python*, which we recommend reading, was 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.

Print is very useful for debugging. It is important to know that you can print out as many items as you want in one line:

```
>>> x = 100
>>> print 'x', x, 'x squared', x*x, 'xiv', 14
x 100 x squared 10000 xiv 14
```

We have also snuck in another data type on you: strings. A string is a sequence of characters. You can create a string using single or double quotes; and access individual elements of strings using indexing.

```
>>> s1 = 'hello world'
>>> s2 = "hello world"
>>> s1 == s2
True
>>> s1[3]
'l'
```

As you can see, indexing refers to the extraction of a particular element of a string, by using square brackets `[i]` where `i` is a number that identifies the location of the character that you wish to extract (*note that the indexing starts with 0*).

Look in the Python documentation for more about strings.

2.3 Control Structures

Python has control structures that are slightly different from those in other languages.

2.3.1 Conditionals

Booleans

Before we talk about conditionals, we need to clarify the Boolean data type. It has values `True` and `False`. Typical expressions that have Boolean values are numerical comparisons:

```
>>> 7 > 8
False
>>> -6 <= 9
True
```

We can also test whether data items are equal to one another. Generally we use `==` to test for equality. It returns `True` if the two objects have equal values. Sometimes, however, we will be interested in knowing whether the two items are the exact same object (in the sense discussed in in Section 3.3). In that case we use `is`:

```
>>> [1, 2] == [1, 2]
True
>>> [1, 2] is [1, 2]
False
>>> a = [1, 2]
>>> b = [1, 2]
>>> c = a
>>> a == b
True
>>> a is b
False
>>> a == c
True
>>> a is c
True
```

Thus, in the examples above, we see that `==` testing can be applied to nested structures, and basically returns true if each of the individual elements is the same. However, `is` testing, especially when applied to nested structures, is more refined, and only returns `True` if the two objects point to exactly the same instance in memory.

In addition, we can combine Boolean values conveniently using `and`, `or`, and `not`:

```
>>> 7 > 8 or 8 > 7
True
>>> not 7 > 8
True
>>> 7 == 7 and 8 > 7
True
```

If

Basic conditional statements have the form:[2]

```
if <booleanExpr>:
```

[2]See the Python documentation for more variations.

```
    <statementT1>
    ...
    <statementTk>
else:
    <statementF1>
    ...
    <statementFn>
```

When the interpreter encounters a conditional statement, it starts by evaluating `<booleanExpr>`, getting either `True` or `False` as a result.[3] If the result is `True`, then it will evaluate `<statementT1>,...,<statementTk>`; if it is `False`, then it will evaluate `<statementF1>,...,<statementFn>`. Crucially, it always evaluates only one set of the statements.

Now, for example, we can implement a procedure that returns the absolute value of its argument.

```
def abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

We could also have written

```
def abs(x):
    if x >= 0:
        result = x
    else:
        result = -x
    return result
```

Python uses the level of indentation of the statements to decide which ones go in the groups of statements governed by the conditionals; so, in the example above, the `return result` statement is evaluated once the conditional is done, no matter which branch of the conditional is evaluated.

For and While

If we want to do some operation or set of operations several times, we can manage the process in several different ways. The most straightforward are `for` and `while` statements (often called `for` and `while` loops).

A `for` loop has the following form:

[3] In fact, Python will let you put any expression in the place of `<booleanExpr>`, and it will treat the values `0`, `0.0`, `[]`, `''`, and `None` as if they were `False` and everything else as `True`.

```
for <var> in <listExpr>:
    <statement1>
    ...
    <statementn>
```

The interpreter starts by evaluating `listExpr`. If it does not yield a list, tuple, or string, an error occurs. If it does yield a list or list-like structure, then the block of statements will, under normal circumstances, be executed one time for every value in that list. At the end, the variable `<var>` will remain bound to the last element of the list (and if it had a useful value before the `for` was evaluated, that value will have been permanently overwritten).

Here is a basic `for` loop:

```
result = 0
for x in [1, 3, 4]:
    result = result + x * x
```

At the end of this execution, `result` will have the value 26, and `x` will have the value 4.

One situation in which the body is not executed once for each value in the list is when a `return` statement is encountered. No matter whether `return` is nested in a loop or not, if it is evaluated it immediately causes a value to be returned from a procedure call. So, for example, we might write a procedure that tests to see if an item is a member of a list, and returns `True` if it is and `False` if it is not, as follows:

```
def member(x, items):
    for i in items:
        if x == i:
            return True
    return False
```

The procedure loops through all of the elements in `items`, and compares them to `x`. As soon as it finds an item `i` that is equal to `x`, it can quit and return the value `True` from the procedure. If it gets all the way through the loop without returning, then we know that `x` is not in the list, and we can return `False`.

Exercise 2.1
Write a procedure that takes a list of numbers, `nums`, and a limit, `limit`, and returns a list which is the shortest prefix of `nums` the sum of whose values is greater than `limit`. Use `for`. Try to avoid using explicit indexing into the list. (Hint: consider the strategy we used in `member`.)

Range

Very frequently, we will want to iterate through a list of integers, often as indices. Python provides a useful procedure, `range`, which returns lists of integers. It can be used in complex ways, but the basic usage is `range(n)`, which returns a list of integers going from 0 up to, but not including, its argument. So `range(3)` returns `[0, 1, 2]`.

Exercise 2.2
Write a procedure that takes `n` as an argument and returns the sum of the squares of the integers from 1 to n-1. It should use `for` and `range`.

Exercise 2.3
What is wrong with this procedure, which is supposed to return `True` if the element `x` occurs in the list `items`, and `False` otherwise?

```
def member (x, items):
    for i in items:
        if x == i:
            return True
        else:
            return False
```

While

You should use `for` whenever you can, because it makes the structure of your loops clear. Sometimes, however, you need to do an operation several times, but you do not know in advance how many times it needs to be done. In such situations, you can use a `while` statement, of the form:

```
while <booleanExpr>:
    <statement1>
    ...
    <statementn>
```

In order to evaluate a `while` statement, the interpreter evaluates `<booleanExpr>`, getting a Boolean value. If the value is `False`, it skips all the statements and evaluation moves on to the next statement in the program. If the value is `True`, then the statements are executed, and the `<booleanExpr>` is evaluated again. If it is `False`, execution of the loop is terminated, and if it is `True`, it goes around again.

It will generally be the case that you initialize a variable before the `while` statement, change that variable in the course of executing the loop, and test some property of that variable in the Boolean expression. Imagine that you wanted to write a procedure that takes an argument `n` and returns the smallest power of 2 that is larger than `n`. You might do it like this:

```
def pow2Smaller(n):
    p = 1
    while p*2 < n:
        p = p*2
    return p
```

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
>>> y[-1]
3
>>> y[-2]
2
>>> len(y)
3
>>> y + [4]
[1, 2, 3, 4]
>>> [4] + y
[4, 1, 2, 3]
>>> [4,5,6] + y
[4, 5, 6, 1, 2, 3]
>>> y
[1, 2, 3]
```

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, like for strings, the indexing starts with 0. Note that you can index elements of a list starting from the initial (or zeroth) one (by using integers), or starting from the last one (by using negative integers).

You can add elements to a list using `'+'`, taking advantage of Python operator overloading. Note that this operation does not change the original list, but makes a new one.

Another useful thing to know about lists is that you can make *slices* of them. A slice of a list is sublist; you can get the basic idea from examples.

```
>>> b = range(10)
>>> b
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> b[1:]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> b[:7]
[0, 1, 2, 3, 4, 5, 6]
>>> b[:-1]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> b[:-2]
[0, 1, 2, 3, 4, 5, 6, 7]
```

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.[4]

First, here is a version in a style you might have learned to write in a Java class (actually, you would have used `for`, but Python does not 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.

Here is a version using Python's `for` loop.

```
def addList2(l):
    sum = 0
    for i in range(len(l)):
```

[4]For any program you will 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 will have to worry about writing a version of a program that runs very quickly, and it might be that in order to make that happen, you will have to write it less cleanly or clearly or briefly. But it is important to have a version that is correct before you worry about getting one that is fast.

```
        sum = sum + l[i]
    return sum
```

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 `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!

This one is even more direct.

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

We do not 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 is called `sum`:

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

List Comprehensions

Python has a very nice built-in facility for doing many iterative operations, called *list comprehensions*. The basic template is

```
[<resultExpr> for <var> in <listExpr> if <conditionExpr>]
```

where `<var>` is a single variable (or a tuple of variables), `<listExpr>` is an expression that evaluates to a list, tuple, or string, and `<resultExpr>` is an expression that may use the variable `<var>`. The `if <conditionExpr>` is optional; if it is present, then only those values of `<var>` for which that expression is `True` are included in the resulting computation.

You can view a list comprehension as a special notation for a particular, very common, class of `for` loops. It is equivalent to the following:

```
*resultVar* = []
for <var> in <listExpr>:
    if <conditionExpr>:
        *resultVar*.append(<resultExpr>)
*resultVar*
```

We used a kind of funny notation `*resultVar*` to indicate that there is some anonymous list that is getting built up during the evaluation of the list comprehension, but we have no real way of accessing it. The result is a list, which is obtained by successively binding `<var>` to elements of the result of evaluating `<listExpr>`, testing to see whether they meet a condition, and if they meet the condition, evaluating `<resultExpr>` and collecting the results into a list.

Whew. It is 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]
>>> [a[0] for a in [['Aditya', 'Mathur'],['Stanley','Kok'],
                    ['Flora','Tsai']]]
['Aditya', 'Stanley', 'Flora']
>>> [a[0]+'!' for a in [['Aditya', 'Mathur'],['Stanley','Kok'],
                        ['Flora','Tsai']]]
['Aditya!', 'Stanley!', 'Flora!']
```

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]
```

Note the use of the `if` conditional here to include only particular values of `x`.

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]
```

You can also use structured assignments in list comprehensions

```
>>> [first for (first, last) in [['Aditya', 'Mathur'],['Stanley','Kok'],
                     ['Flora','Tsai']]]
['Aditya', 'Stanley', 'Flora']
>>> [first+last for (first, last) in [['Aditya', 'Mathur'],['Stanley','Kok'],
                     ['Flora','Tsai']]]
['AdityaMathur', 'StanleyKok', 'FloraTsai']
```

Another built-in function that is useful with list comprehensions is `zip`. Here are some examples of how it works:

```
> zip([1, 2, 3],[4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
> zip([1,2], [3, 4], [5, 6])
[(1, 3, 5), (2, 4, 6)]
```

Here is an example of using `zip` with a list comprehension:

```
>>> [first+last for (first, last) in zip(['Aditya', 'Stanley', 'Flora'],
                                          ['Mathur','Kok','Tsai'])]
['AdityaMathur', 'StanleyKok', 'FloraTsai']
```

Note that this last example is very different from this one:

```
>>> [first+last for first in ['Aditya', 'Stanley', 'Flora'] \
                for last in ['Mathur','Kok','Tsai']]
['AdityaMathur', 'AdityaKok', 'AdityaTsai', 'StanleyMathur', 'StanleyKok',
'StanleyTsai', 'FloraMathur', 'FloraKok', 'FloraTsai']
```

Nested list comprehensions behave like nested `for` loops, the expression in the list comprehension is evaluated for every combination of the values of the variables.

2.4 Common Errors and Messages

Here are some common Python errors and error messages to watch out for. Please let us know if you have any favorite additions for this list.

- A complaint about `NoneType` often means you forgot a return.

```
def plus1 (x):
    x + 1
>>> y = plus1(x)
>>> plus1(x) + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

- Weird results from math can result from integer division

```
>>> 3/ 9
0
```

- "Unsubscriptable object" means you are trying to get an element out of something that isn't a dictionary, list, or tuple.

```
>>> x = 1
>>> x[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is unsubscriptable
```

- "Object is not callable" means you are trying to use something that isn't a procedure or method as if it were.

```
>>> x = 1
>>> x(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

- "List index out of range" means you are trying to read or write an element of a list that is not present.

```
>>> a = range(5)
>>> a
[0, 1, 2, 3, 4]
>>> a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- "Maximum recursion depth exceeded" means you have a recursive procedure that is nested *very* deeply or your base case is not being reached due to a bug.

```
def fizz(x):
    return fizz(x - 1)
>>> fizz(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fizz
  File "<stdin>", line 2, in fizz
...
  File "<stdin>", line 2, in fizz
RuntimeError: maximum recursion depth exceeded
```

- "Key Error" means that you are trying to look up an element in a dictionary that is not present.

```
>>> d = {'a':7, 'b':8}
>>> d['c']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
```

- Another common error is forgetting the `self` before calling a method. This generates the same error that you would get if you tried to call a function that wasn't defined at all.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "V2.py", line 22, in __add__
    return add(v)
NameError: global name 'add' is not defined
```

2.5 Python Style

Software engineering courses often provide very rigid guidelines on the style of programming, specifying the appropriate amount of indentation, or what to capitalize, or whether to use underscores in variable names. Those things can be useful for uniformity and readability of code, especially when a lot of people are working on a project. But they are mostly arbitrary: a style is chosen for consistency and according to some person's aesthetic preferences.

There are other matters of style that seem, to us, to be more fundamental, because they directly affect the readability or efficiency of the code.

- **Avoid recalculation of the same value.** You should compute it once and assign it to a variable instead; otherwise, if you have a bug in the calculation (or you want to change the program), you will have to change it multiple times. It is also inefficient.
- **Avoid repetition of a pattern of computation.** You should use a function instead, again to avoid having to change or debug the same basic code multiple times.

- **Avoid numeric indexing.** You should use destructuring if possible, since it is much easier to read the code and therefore easier to get right and to modify later.
- **Avoid excessive numeric constants.** You should name the constants, since it is much easier to read the code and therefore easier to get right and to modify later.

Here are some examples of simple procedures that exhibit various flaws. We'll talk about what makes them problematic.

2.5.1 Normalize a Vector

Let's imagine we want to normalize a vector of three values; that is to compute a new vector of three values, such that its length is 1. Here is our first attempt; it is a procedure that takes as input a list of three numbers, and returns a list of three numbers:

```
def normalize3(v):
    return [v[0]/math.sqrt(v[0]**2+v[1]**2+v[2]**2),
            v[1]/math.sqrt(v[0]**2+v[1]**2+v[2]**2),
            v[2]/math.sqrt(v[0]**2+v[1]**2+v[2]**2)]
```

This is correct, but it looks pretty complicated. Let's start by noticing that we're recalculating the denominator three times, and instead save the value in a variable.

```
def normalize3(v):
    magv = math.sqrt(v[0]**2+v[1]**2+v[2]**2)
    return [v[0]/magv,v[1]/magv,v[2]/magv]
```

Now, we can see a repeated pattern, of going through and dividing each element by `magv`. Also, we observe that the computation of the magnitude of a vector is a useful and understandable operation in its own right, and should probably be put in its own procedure. That leads to this procedure:

```
def mag(v):
    return math.sqrt(sum([vi**2 for vi in v]))

def normalize3(v):
    return [vi/mag(v) for vi in v]
```

This is especially nice, because now, in fact, it applies not just to vectors of length three. So, it's both shorter and more general than what we started with. But, one of our original problems has snuck back in: we're recomputing `mag(v)` once for each element of the vector. So, finally, here's a version that we're very happy with:[5]

[5]Note that there is still something for someone to be unhappy with here: the use of a list comprehension means that we're creating a new list, which we are just going to sum up; that's somewhat less efficient than adding the values up in a loop. However, as we said at the outset, for almost every program, clarity matters more than efficiency. And once you have something that's clear and correct, you can selectively make the parts that are executed frequently more efficient.

```
def mag(v):
    return math.sqrt(sum([vi**2 for vi in v]))

def normalize3(v):
    magv = mag(v)
    return [vi/magv for vi in v]
```

2.5.2 Perimeter of a Polygon

Now, let's consider the problem of computing the length of the perimeter of a polygon. The input is a list of vertices, encoded as a list of lists of two numbers (such as `[[1, 2], [3.4, 7.6], [-4.4, 3]]`). Here is our first attempt:

```
def perim(vertices):
    result = 0
    for i in range(len(vertices)-1):
        result = result + math.sqrt((vertices[i][0]-vertices[i+1][0])**2 + \
                                    (vertices[i][1]-vertices[i+1][1])**2)
    return result + math.sqrt((vertices[-1][0]-vertices[0][0])**2 + \
                              (vertices[-1][1]-vertices[0][1])**2)
```

Again, this works, but it ain't pretty. The main problem is that someone reading the code doesn't immediately see what all that subtraction and squaring is about. We can fix this by defining another procedure:

```
def perim(vertices):
    result = 0
    for i in range(len(vertices)-1):
        result = result + pointDist(vertices[i],vertices[i+1])
    return result + pointDist(vertices[-1],vertices[0])

def pointDist(p1,p2):
    return math.sqrt(sum([(p1[i] - p2[i])**2 for i in range(len(p1))]))
```

Now, we've defined a new procedure `pointDist`, which computes the Euclidean distance between two points. And, in fact, we've written it generally enough to work on points of any dimension (not just two). Just for fun, here's another way to compute the distance, which some people would prefer and others would not.

```
def pointDist(p1,p2):
    return math.sqrt(sum([(c1 - c2)**2 for (c1, c2) in zip(p1, p2)]))
```

2.5.3 Bank Transfer

What if we have two values, representing bank accounts, and want to transfer an amount of money `amt` between them? Assume that a bank account is represented as a list of values, such as `['Alyssa', 8300343.03, 0.05]`, meaning that `'Alyssa'` has a bank balance of $8,300,343.03, and has to pay a 5-cent fee for every bank transaction. We might write this procedure as follows. It moves the amount from one balance to the other, and subtracts the transaction fee from each account.

```
def transfer(a1, a2, amt):
    a1[1] = a1[1] - amt - a1[2]
    a2[1] = a2[1] + amt - a2[2]
```

To understand what it's doing, you really have to read the code at a detailed level. Furthermore, it's easy to get the variable names and subscripts wrong.

Here's another version that abstracts away the common idea of a deposit (which can be positive or negative) into a procedure, and uses it twice:

```
def transfer(a1, a2, amt):
    deposit(a1, -amt)
    deposit(a2, amt)

def deposit(a, amt):
    a[1] = a[1] + amt - a[2]
```

Now, `transfer` looks pretty clear, but `deposit` could still use some work. In particular, the use of numeric indices to get the components out of the bank account definition is a bit cryptic (and easy to get wrong).[6]

```
def deposit(a, amt):
    (name, balance, fee) = a
    a[1] = balance + amt - fee
```

Here, we've used a destructuring assignment statement to give names to the components of the account. Unfortunately, when we want to change an element of the list representing the account, we still have to index it explicitly. Given that we have to use explicit indices, this approach in which we name them might be better.

```
acctName = 0
acctBalance = 1
acctFee = 2
def deposit(a, amt):
    a[acctBalance] = a[acctBalance] + amt - a[acctFee]
```

[6]We'll see other approaches to this when we start to look at object-oriented programming. But it's important to apply basic principles of naming and clarity no matter whether you're using assembly language or Java.

Strive, in your programming, to make your code as simple, clear, and direct as possible. Occasionally, the simple and clear approach will be too inefficient, and you'll have to do something more complicated. In such cases, you should still start with something clear and simple, and in the end, you can use it as documentation.

Chapter 3

Object-Oriented Programming

Object-oriented programming is a popular way of organizing programs, which groups together data with the procedures that operate on them, thus facilitating some kinds of modularity and abstraction. In the context of our PCAP framework, object-oriented programming will give us methods for capturing common patterns in data and the procedures that operate on that data, via classes, generic functions, and inheritance.

In this chapter, we will try to develop a deep understanding of object-oriented programming by working through the mechanism by which an interpreter evaluates a computer program. The first part of the chapter will focus on interpretation of typical expressions, starting from the simplest single-statement programs and working up through list structures and procedures. Many of the observations made through this process apply to styles of program organization as well as object-oriented programming. Once we understand how an interpreter evaluates standard expressions, we will move to objects and classes. Although we use Python as an example, the discussion in this chapter is intended to be illustrative of principles of computer languages, more generally.

In many computer languages, including Python, programs are understood and executed by a computer program called an *interpreter*. Interpreters are surprisingly simple: the rules defining the meaning or *semantics* of a programming language are typically short and compact; and the interpreter basically encodes these rules and applies them to any legal expression in the language. The enormous richness and complexity of computer programs come from the composition of primitive elements with simple rules. The interpreter, in essence, defines the semantics of the language by capturing the rules governing the value or behavior of program primitives, and of what it means to combine the primitives in various ways. We will study the meaning of computer programs by understanding how the interpreter operates on them.

An interpreter is made up of four pieces:

- The *reader* or *tokenizer* takes as input a string of characters and divides them into *tokens*, which are numbers (like `-3.42`), words (like `while` or `a`), and special characters (like `:`).
- The *parser* takes as input the string of tokens and understands them as constructs

in the programming language, such as while loops, procedure definitions, or return statements.

- The *evaluator* (which is also sometimes called the *interpreter*, as well) has the really interesting job of determining the value and effects of the program that you ask it to interpret.
- The *printer* takes the value returned by the evaluator and prints it out for the user to see.

Programs should never be a mystery to you: you can learn the simple semantic rules of the language and, if necessary, simulate what the interpreter would do, in order to understand *any* computer program that you are facing. Of course, in general, one does not want to work through the tedious process of simulating the interpreter, but this foundation of understanding the interpreter's process enables you to reason about the evaluation of any program.

3.1 Primitives, Composition, Abstraction, and Patterns

We will start by thinking about how the PCAP framework applies to computer programs, in general. We can do this by filling in Table 3.1, exploring the PCAP ideas in data, procedures, and objects.

Data

The primitive data items in most programming languages are things like integers, floating point numbers, and strings. We can combine these into data structures (we discuss some basic Python data structures in Section 3.3) such as lists, arrays, dictionaries and records. Making a data structure allows us, at the most basic level, to think of a collection of primitive data elements as if it were one thing, freeing us from details. Sometimes, we just want to think of a collection of data, not in terms of its underlying representation, but in terms of what it represents. So, we might want to think of a set of objects, or a family tree, without worrying whether it is an array or a list in its basic representation. *Abstract data types* provide a way of abstracting away from representational details and allowing us to focus on what the data really means.

Procedures

The primitive procedures of a language are things like built-in numeric operations and basic list operations. We can combine these using the facilities of the language, such as `if` and `while`, or by using function composition (`f(g(x))`). If we want to abstract away from the details of how a particular computation is done, we can define a new function; defining a function allows us to use it for computational jobs without thinking about the details of how those computational jobs get done. You can think of this process as essentially creating a new primitive, which we can then use while ignoring the details of how it is constructed. One way to capture common patterns of abstraction in procedures is to abstract over procedures themselves, with higher-order procedures, which we discuss in detail in Section 3.4.6.

Objects

Object-oriented programming provides a number of methods of abstraction and pattern capture in both data and procedures. At the most basic level, objects can be used as records, combining together primitive data elements. More generally, they provide strategies for jointly abstracting a data representation and the procedures that work on it. The features of *inheritance* and *polymorphism* are particularly important, and we will discuss them in detail later in this chapter.

	Procedures	Data
Primitives	`+`, `*`, `==`	numbers, strings
Means of **combination**	`if`, `while`, `f(g(x))`	lists, dictionaries, objects
Means of **abstraction**	`def`	classes
Means of capturing **patterns**	higher-order procedures	generic functions, inheritance

Table 3.1: Primitives, combination, abstraction, patterns framework for computer programs

3.2 Expressions and Assignment

We can think of most computer programs as performing some sort of transformation on data. Our program might take as input the exam scores of everyone in the class and generate the average score as output. Or, in a transducer model, we can think about writing the program that takes the current memory state of the transducer and an input, and computes a new memory state and output. (You will learn more about this in subsequent chapters.)

To represent data in a computer, we have to encode it, ultimately as sequences of binary digits (0s and 1s). The memory of a computer is divided into 'words', which typically hold 32 or 64 bits; a word can be used to store a number, one or several characters, or a pointer to (i.e., the address of) another memory location.

A computer program, at the lowest level, is a set of primitive instructions, also encoded into bits and stored in the words of the computer's memory. These instructions specify operations to be performed on the data (and sometimes the program itself) that are stored in the computer's memory. In this class, we will not work at the level of these low-level instructions. A high-level programming language such as Python let us abstract away from these details. But it is important to have an abstract mental model of what is going on within the computer.

3.2.1 Simple Expressions

A cornerstone of a programming language is the ability to evaluate expressions. We will start here with arithmetic expressions, just to get the idea. An expression consists of a sequence of 'tokens' (words, special symbols, or numerals) that represent the application of operators to data elements. Each expression has a value, which can be computed recursively by evaluating primitive expressions, and then using standard rules to combine their values to get new values.

Numerals, such as `6` or `-3.7` are primitive expressions, whose values are numeric constants. Their values can be *integers*, within some fixed range dictated by the programming language, or *floating point numbers*. Floating point numbers are used to represent non-integer values, but they are different, in many important ways, from the real numbers. There are infinitely many real numbers within a finite interval, but only finitely many floating-point numbers exist at all (because they all must be representable in a fixed number of bits).

We will illustrate the evaluation of expressions in Python by showing short transcripts of interactive sessions with the *Python shell*. The shell is a computer program that

- Prompts the user for an expression, by typing `>>>`,
- Reads what the user types in, and converts it into a set of tokens,
- Parses the tokens into a data structure representing the syntax of the expression,
- Evaluates the parsed expression using an interpreter, and
- Prints out the resulting value

So, for example, we might have this interaction with Python:

```
>>> 2 + 3
5
>>> (3 * 8) - 2
22
>>> ((3 * 8) - 2) / 11
2
>>> 2.0
2.0
>>> 0.1
0.10000000000000001
>>> 1.0 / 3.0
0.33333333333333331
>>> 1 / 3
0
```

There are a couple of things to observe here. First, we can see how floating point numbers only approximately represent real numbers: when we type in `0.1`, the closest Python can come to it in floating point is `0.10000000000000001`. The last interaction is particularly troubling: it seems like the value of the expression `1 / 3` should be something like `0.33333`. However, in Python, if both operands to the `/` operator are integers, then it will perform an integer division, truncating any remainder.[1]

These expressions can be arbitrarily deeply nested combinations of primitives. The rules used for evaluation are essentially the same as the ones you learned in school; the interpreter proceeds by applying the operations in precedence order[2], evaluating sub-expressions to get new values, and then evaluating the expressions those values participate in, until a single value results.

3.2.2 Variables

We cannot go very far without *variables*. A variable is a name that we can *bind* to have a particular value and then later use in an expression. When a variable is encountered in an expression, it is evaluated by looking to see to what value it is bound.

An interpreter keeps track of which variables are bound to what values in *binding environments*. An environment specifies a mapping between variable names and values. The values can be integers, floating-point numbers, characters, or pointers to more complex entities such as procedures or larger collections of data.

Here is an example binding environment:

b	3
x	2.2
foo	-1012

Each row represents a binding: the entry in the first column is the variable name and the entry in the second column is the value it to which it is bound.

When you start up the Python shell, you immediately start interacting with a local binding environment. You can add a binding or change an existing binding by evaluating an assignment statement of the form:

```
<var> = <expr>
```

where `<var>` is a variable name (a string of letters or digits or the character _, not starting with a digit) and `<expr>` is a Python expression.[3]

Expressions are always evaluated in some environment.

[1]This behavior will no longer be the default in Python 3.0.

[2]Please Excuse My Dear Aunt Sally (Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction)

[3]When we want to talk about the abstract form or `syntax` of a programming language construct, we will often use *meta-variables*, written with angle brackets, like `<var>`. This is meant to signify that `<var>` could be any Python variable name, for example.

We might have the following interaction in a fresh Python shell:

```
>>> a = 3
>>> a
3
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>>
```

We started by assigning the variable `a` to have the value `3`. That added a binding for `a` to the local environment.

Next, we evaluated the expression `a`. The value of an expression with one or more variable names in it cannot be determined unless we know with respect to what environment it is being evaluated. Thus, we will always speak of evaluating expressions *in* an environment. During the process of evaluating an expression in some environment E, if the interpreter comes to a variable, it looks up that variable in E: if E contains a binding for the variable, then the associated value is returned; if it does not, then an error is generated. In the Python shell interaction above, we can see that the interpreter was able to find a binding for `a` and return a value, but it was not able to find a binding for `b`.

Why do we bother defining values for variables? They allow us to re-use an intermediate value in a computation. We might want to compute a formula in two steps, as in:

```
>>> c = 952**4
>>> c**2 + c / 2.0
6.7467650588636822e+23
```

They will also play a crucial role in abstraction and the definition of procedures. By giving a name to a value, we can isolate the use of that value in other computations, so that if we decide to change the value, we only have to change the definition (and not change a value several places in the code).

It is fine to reassign the value of a variable; although we use the equality symbol `=` to stand for assignment, we are not making a mathematical statement of equality. So, for example, we can write:

```
>>> a = 3
>>> a = a + 1
>>> a
4
```

Exercise 3.1
What is the result of evaluating this sequence of assignment statements and the last expression? Determine this by hand-simulating the Python interpreter. Draw an environment and update the stored values as you work through this example.

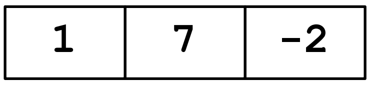
```
>>> a = 3
>>> b = a
>>> a = 4
>>> b
```

3.3 Structured Data

We will often want to work with large collections of data. Rather than giving each number its own name, we want to organize the data into natural structures: grocery lists, matrices, sets of employee medical records. In this section, we will explore a simple but enormously useful and flexible data structure, which is conveniently built into Python: the list. The precise details of how lists are represented inside a computer vary from language to language. We will adopt an abstract model in which we think of a list as an ordered sequence of memory locations that contain values. So, for example, in Python, we can express a list of three integers as:

```
>>> [1, 7, -2]
[1, 7, -2]
```

which we will draw in an abstract memory diagram as:



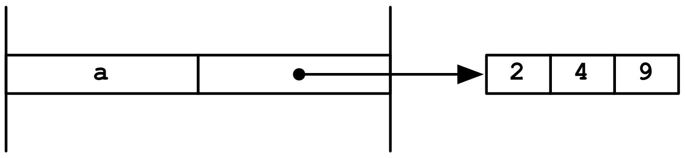
We can assign a list to a variable:

```
>>> a = [2, 4, 9]
```

A binding environment associates a name with a single fixed-size data item. So, if we want to associate a name with a complex structure, we associate the name directly with a 'pointer' to (actually, the memory address of) the structure. So we can think of `a` as being bound to a 'pointer' to the list:

Now that we have lists, we have some new kinds of expressions, which let us extract components of a list by specifying their indices. An index of 0 corresponds to the first element of a list. An index of -1 corresponds to the last element (no matter how many elements there are).[4] So, if `a` is bound as above, then we would have:

[4]See the Python tutorial for much more on list indexing.



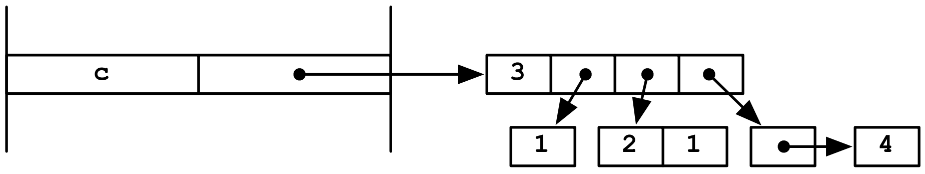
```
>>> a[0]
2
>>> a[2]
9
>>> a[-1]
9
>>> a[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Note that if we attempt to access an element of the list that is not present (in this case, the fourth element of a three-element list), then an error is generated.

Lists can be nested inside one another. The Python expression:

```
>>> c = [3, [1], [2, 1], [[4]]]
```

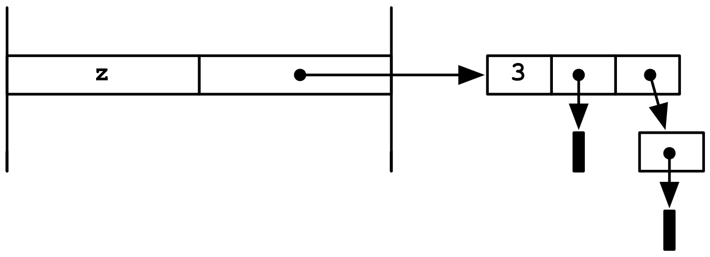
creates a list that looks, in memory, like this:



It is also possible to have an empty list, which is written in Python as `[]`. We will draw it in our memory diagrams as a small black box. So, for example, this list

```
>>> z = [3, [], [[]]]
```

looks like this in memory:



Python has a useful function, `len`, which takes a list as an argument and returns its length. It does not look inside the elements of the list—it just returns the number of elements at the top level of structure. So, we have

```
>>> len([1, 2, 3])
3
>>> len([[1, 2, 3]])
1
```

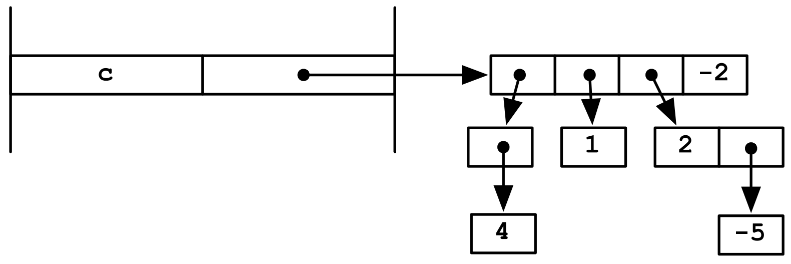
Exercise 3.2
Draw a diagram of the binding environment and memory structure after the following statement has been evaluated:

```
a = [[1], 2, [3, 4]]
```

Exercise 3.3
Draw a diagram of the binding environment and memory structure after the following statement has been evaluated:

```
a = [[[]]]
```

Exercise 3.4
Give a Python statement which, when evaluated, would give rise to this memory structure:



What is the value, in this environment, of the following expressions:

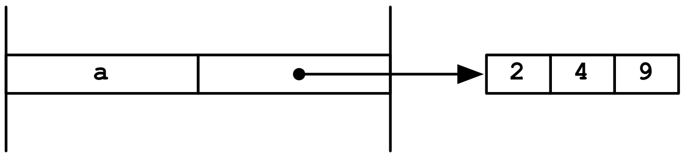
- `c[1]`
- `c[-1]`
- `c[2][1]`

3.3.1 List Mutation and Shared Structure

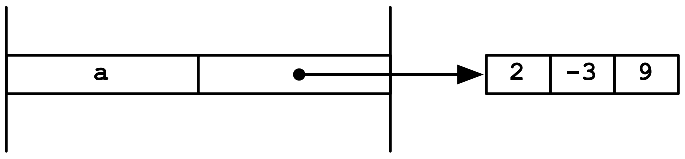
Lists are *mutable* data structures, which means that we can actually change the values stored in their elements. We do this by using element-selection expressions, like `a[1]` on the left-hand side of an assignment statement. So, the assignment

```
a[1] = -3
```

assigns the second element of `a` to be `-3`. In more detail, the left-hand side of this expression evaluates to a pointer to a specific location in memory (just as `a`'s value is a pointer to a location in memory); then the assignment statement changes the value stored there by inserting the value of the right-hand side of the expression. If that statement were evaluated in this environment,



then the resulting environment would be:



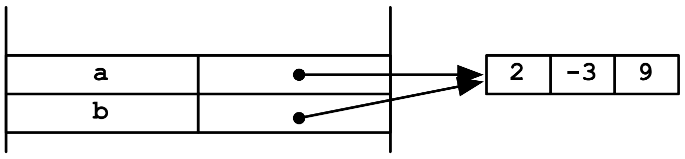
We have permanently changed the list named `a`.

In this section, we will explore the consequences of the mutability of lists; programs that change list structure can become *very* confusing, but you can always work your way through what is happening by drawing out the memory diagrams.

Continuing the previous example, let us remember that `a` is bound directly to a pointer to a list (or a sequence of memory cells), and think about what happens if we do:

```
>>> b = a
```

Now, `a` and `b` are both names for *the same list structure*, resulting in a memory diagram like this:



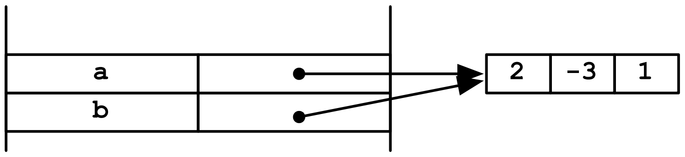
Now, we can reference parts of the list through `b`, and even change the list structure that way:

```
>>> b[0]
2
>>> b[2] = 1
```

Notice that, because `a` and `b` point to the same list, changing `b` changes `a`!

```
>>> a
[2, -3, 1]
```

Here is the memory picture now:

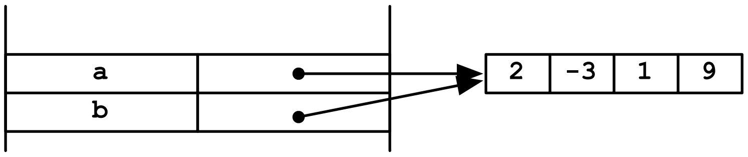


This situation is called *aliasing*: the name `b` has become an alias for `a`. Aliasing can be useful, but it can also cause problems, because you might inadvertently change `b` (by passing it into a procedure that changes one of its structured arguments, for example) when it is important to you to keep `a` unmodified.

Another important way to change a list is to add or delete elements. We will demonstrate adding elements to the end of a list, but see the Python documentation for more operations on lists. This statement

```
>>> a.append(9)
```

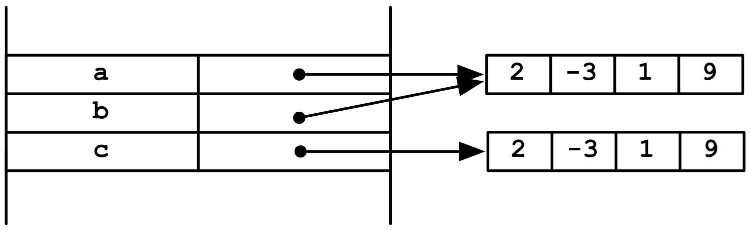
causes a new element to be added to the end of the list name `a`. The resulting memory state is:



As before, because `a` and `b` are names for the same list (i.e., they point to the same memory sequence), `b` is changed too. This is a side effect of the aliasing between `a` and `b`:

```
>>> b
[2, -3, 1, 9]
```

Often, it will be important to make a fresh copy of a list so that you can change it without affecting the original one. Here are two equivalent ways to make a copy (use whichever one you can remember):



```
>>> c = list(a)
>>> c = a[:]
```

Here is a picture of the memory at this point:

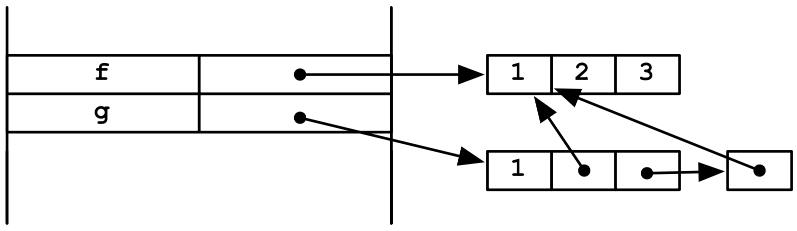
Now, if we change an element of `c`, it does not affect `a` (or `b`!):

```
>>> c[0] = 100
>>> c
[100, -3, 1, 9]
>>> a
[2, -3, 1, 9]
```

We can make crazy lists that share structure within a single list:

```
>>> f = [1, 2, 3]
>>> g = [1, f, [f]]
>>> g
[1, [1, 2, 3], [[1, 2, 3]]]
```

which results in this memory structure:

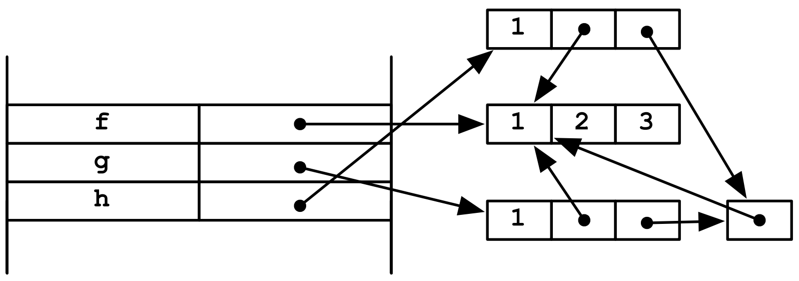


If you want to add an element to a list and get a new copy at the same time, you can do

```
>>> a + [1]
```

The `+` operator makes a new list that contains the elements of both of its arguments, but does not share any top-level structure. All of our methods of copying only work reliably if your lists do not contain other lists, because it **only copies one level of list structure**. So, for example, if we did:

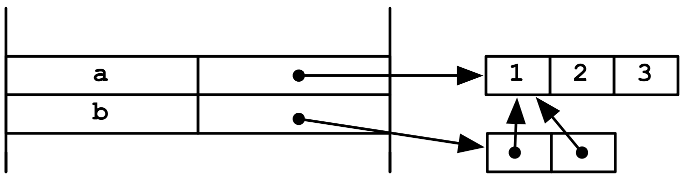
```
>>> h = list(g)
```



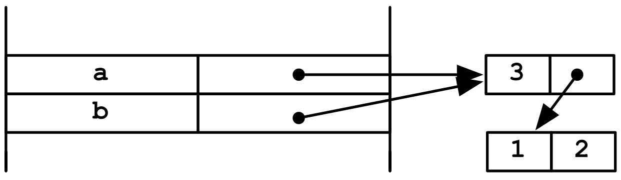
we would end up with this picture:

It is clear that if we were to change `f`, it would change `h`, so this is not a completely new copy. If you need to copy deep structures, that is, to make a copy not only of the top level list structure, but of the structures of any lists that list contains, and the lists those lists contain, etc., you will need to use the Python `copy.deepcopy` procedure.

Exercise 3.5
Give a sequence of Python statements which, when evaluated, would give rise to this memory structure:



Exercise 3.6
Give a sequence of Python statements which, when evaluated, would give rise to this memory structure:



Exercise 3.7
Show the memory structure after this sequence of expressions.

```
>>> a = [1, 2, 3]
>>> b = [a, a]
>>> a.append(100)
```

What will be the value of `b` at this point?

Exercise 3.8
Show the memory structure after this sequence of expressions.

```
>>> a = [5, 6]
>>> b = [1, 2]
>>> c = b + a
```

Exercise 3.9
Show the memory structure after this sequence of expressions.

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

What will be the value of `a` at this point?

3.3.2 Tuples and Strings

Python has two more list-like data types that are important to understand.

A *tuple* is a structure that is like a list, but is *not* mutable. You can make new tuples, but you cannot change the contents of a tuple or add elements to it. A tuple is typically written like a list, but with round parentheses instead of square ones:

```
>>> a = (1, 2, 3)
```

In fact, it is the commas and not the parentheses that matter here. So, you can write

```
>>> a = 1, 2, 3
>>> a
(1, 2, 3)
```

and still get a tuple. The only tricky thing about tuples is making a tuple with a single element. We could try

```
>>> a = (1)
>>> a
1
```

but it does not work, because in the expression `(1)` the parentheses are playing the standard grouping role (and, in fact, because parentheses do not make tuples). So, to make a tuple with a single element, we have to use a comma:

```
>>> a = 1,
>>> a
(1,)
```

This is a little inelegant, but that's the way it is.

Tuples will be important in contexts where we are using structured objects as 'keys', that is, to index into another data structure, and where inconsistencies would occur if those keys could be changed.

An important special kind of tuple is a *string*. A string can almost be thought of as a tuple of characters. The details of what constitutes a character and how they are encoded is complicated, because modern character sets include characters from nearly all the world's languages. We will stick to the characters we can type easily on our keyboards. In Python, you can write a string with either single or double quotes: `'abc'` or `"abc"`. You can select parts of it as you would a list:

```
>>> s = 'abc'
>>> s[0]
'a'
>>> s[-1]
'c'
```

The strange thing about this is that `s` is a string, and because Python has no special data type to represent a single character, `s[0]` is also a string.

We will frequently use `+` to concatenate two existing strings to make a new one:

```
>>> to = 'Jody'
>>> fromP = 'Robin'
>>> letter = 'Dear ' + to + ",\n It's over.\n" + fromP
>>> print letter
Dear Jody,
 It's over.
Robin
```

As well as using `+` to concatenate strings, this example illustrates several other small but important points:

- You can put a single quote inside a string that is delimited by double-quote characters (and vice versa).
- If you want a new line in your string, you can write `\n`. Or, if you delimit your string with a *triple* quote, it can go over multiple lines.
- The `print` statement can be used to print out results in your program.
- Python, like most other programming languages, has some *reserved words* that have special meaning and cannot be used as variables. In this case, we wanted to use `from`, but that has a special meaning to Python, so we used `fromP` instead.

Structured assignment

Once we have lists and tuples, we can use a nice trick in assignment statements, based on the packing and unpacking of tuples.

```
>>> 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
```

3.4 Procedures

Procedures are computer-program constructs that let us capture common patterns of computation by:

- Gathering together sequences of statements
- Abstracting away from particular data items on which they operate.

Here is a procedure definition,[5] and its use:

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

>>> square(6)
36
>>> square(2 - square(2))
4
```

We will work through, in detail, what happens when the interpreter evaluates a procedure definition, and then the application of that procedure.

3.4.1 Definition

A procedure definition has the abstract form:

```
def <name>(<fp1>, ..., <fpn>):
    <statement1>
    ...
    <statementk>
```

There are essentially three parts:

- `<name>` is a name for the procedure, with the same restrictions as a variable name;
- `<fp1>, ..., <fpn>` is a list of *formal parameters*, which will stand for the data items on which this procedure will operate; and
- `<statement1>, ..., <statementk>`, known as the *body of the procedure*, is a list of Python statements (right now, we know about assignment statements, print statements, and basic expressions, but there will be more.)

When we evaluate a procedure definition in an environment,[6] E, Python does two things:

1 Makes a procedure object[7] that contains the formal parameters, the body of the procedure, and a pointer to E; and then

[5]In the code displayed in these notes, we will show procedures being defined and then used, as if the definitions were happening in the Python shell (but without the prompts). In fact, you should not type procedure definitions into the shell, because if you make a mistake, you will have to re-type the whole thing, and because multi-line objects are not handled very well. Instead, type your procedure definitions into a file in Idle, and then test them by 'running' the file in Idle (which will actually evaluate all of the expressions in your file) and then evaluating test expressions in Idle's shell.

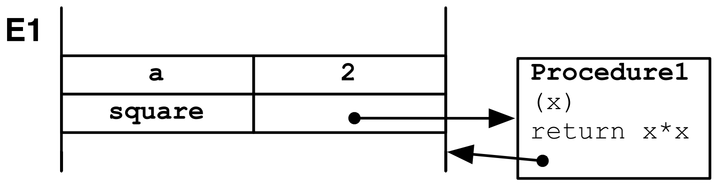
[6]Remember that every expression is always evaluated with respect to some environment.

[7]In our memory diagrams, we will show the procedure object as a box with the word **Procedure<N>**, where **N** is some integer, at the top; we give the procedure objects these numbers so we can refer to them easily in the text.

2 Binds the name to have this procedure as its value.

Here is an example of an environment after we have evaluated

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



Note how the construct to which `square` points is a procedure object, with a list of formal parameters, a body, and a pointer to its environment.

3.4.2 Procedure calls

When you evaluate an expression of the form

```
<expr0>(<expr1>, ..., <exprn>)
```

the Python interpreter treats this as a procedure call. It will be easier to talk about a specific case of calling a procedure, so we will illustrate with the example

```
>>> square(a + 3)
```

evaluated in environment $E1$. Here are the steps:

1 The expression that determines the procedure (`<expr0>`) is evaluated. In this case, we evaluate `square` in $E1$ and get `Procedure1`.

2 The expressions that determine the arguments (`<expr1>, ..., <exprn>`) are evaluated. In this case, we evaluate `a + 3` in $E1$ and get 5.

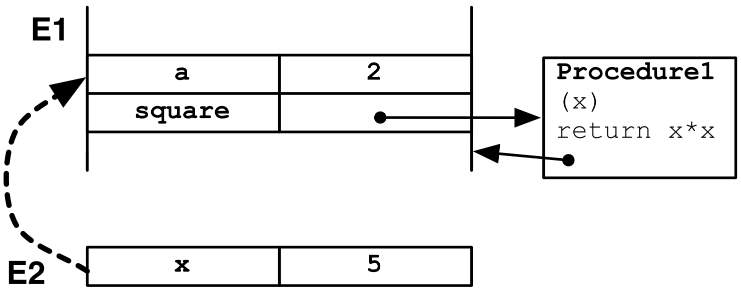
3 A *new environment* (in this case $E2$) is created, which:

   4 binds the formal parameters of the procedure (in this case `x`) to the values of its arguments (in this case, 5); and

   5 has as its parent the environment in which the procedure was defined (we find a pointer to this environment stored in the procedure; in this case $E1$ is its *parent*).

   At this point, our memory looks like this:

   The dotted line between $E2$ and $E1$ is intended to indicate that $E1$ is the *parent environment* of $E2$.



- The statements of the procedure body are evaluated in the new environment until either a `return` statement or the end of the list of statements is reached. If a `return` statement is evaluated, the expression after the `return` is evaluated and its value is returned as the value of the procedure-call expression. Otherwise, the procedure has no return value, and the expression has the special Python value `None`.

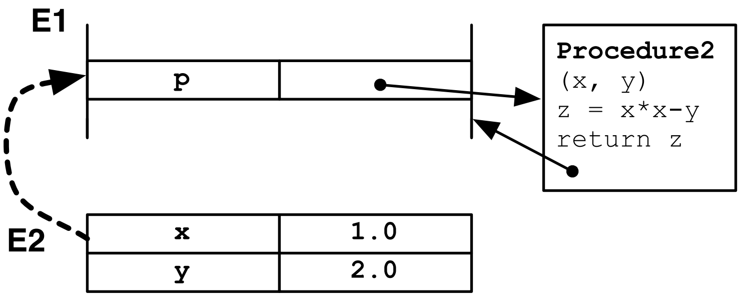
  In our example, the only statement in the body is a return statement. So, we evaluate the expression `x * x` in $E2$, obtaining a value of 25. That value is returned as the value of the entire procedure-call expression `square(a + 3)`.

Worked example 1

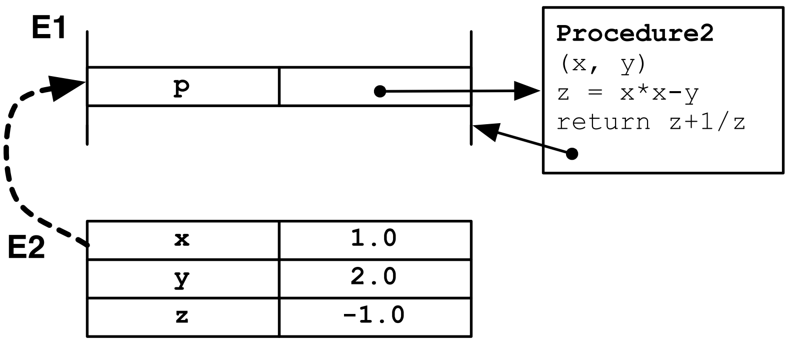
Let's examine what happens when we evaluate the following Python code:

```
def p(x, y):
    z = x*x - y
    return z + 1/z

>>> p(1.0, 2.0)
-2.0
```

Here is a picture of the calling environment ($E1$) and the procedure-call environment ($E2$) just before the body of the procedure is evaluated:



After evaluating `z = x*x - y` in $E2$, we have:

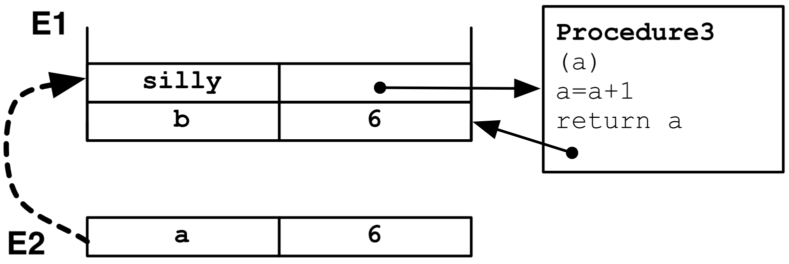


Finally, we evaluate `z + 1/z` in $E2$ and get -2.0, which we return.

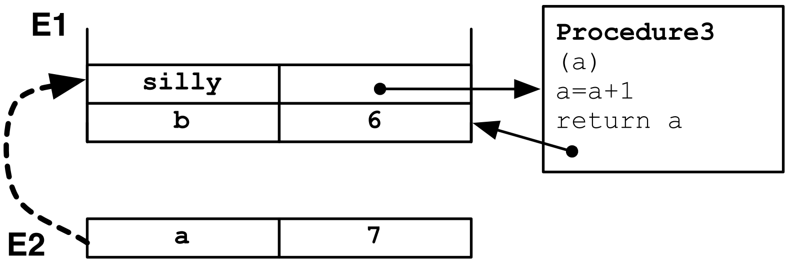
Worked example 2 Here is another example:

```
def silly(a):
    a = a + 1
    return a

>>> b = 6
>>> silly(b)
7
```

Here is a picture of the calling environment ($E1$) and the procedure-call environment ($E2$) just before the body of the procedure is evaluated:



After evaluating `a = a + 1` in $E2$, we have:



Finally, we evaluate `a` in $E2$ and get 7, which we return.

Convince yourself that the execution below works fine, and notice that having a binding for `a` in $E2$ means that we leave the binding for `a` in $E1$ unmolested:

```
>>> a = 2
>>> silly(a)
3
>>> a
2
```

Exercise 3.10
Draw the environment diagram at the time the statement with the arrow is being executed:

```
def fizz(x, y):
    p = x + y
    q = p*p
    return q + x    #   <-----------

>>> fizz(2, 3)
```

Exercise 3.11
Draw the environment diagram at the time the statement with the arrow is being executed:

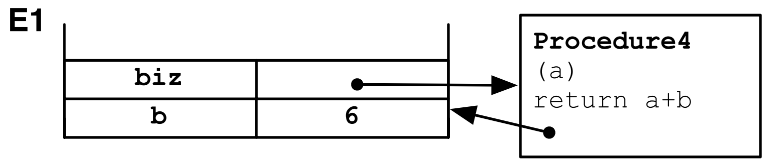
```
def fuzz(a, b):
    return a + b    #   <----------

def buzz(a):
    return fuzz(a, square(a))

>>> buzz(2)
```

3.4.3 Non-Local References

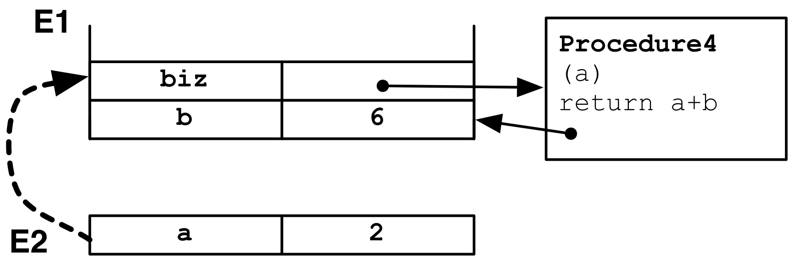
So far, whenever we needed to evaluate a variable, there was a binding for that variable in the 'local' environment (the environment in which we were evaluating the expression). But consider this example:

```
def biz(a):
    return a + b

>>> b = 6
>>> biz(2)
```

This actually works fine, and returns 8. Let's see why. Here is the environment, $E1$, in which the Python shell is working, after we execute the `b = 6` statement:



Now, when we evaluate `biz(2)` in $E1$, we make a new environment, $E2$, which binds `a` to 2 and *points to $E1$ as its parent environment.*



We need to elaborate, slightly, how it is that a variable v is evaluated in an environment E:

- We look to see if there is a binding for v in E; if so, we stop and return it.
- If not, we evaluate v in the parent environment of E. If E has no parent, we generate an error.

It is important to appreciate that this process will continue up an arbitrarily long chain of environments and their parents until either a binding for v is found or there are no more environments to examine.

So, in our case, we will evaluate the expression `a + b` in environment $E2$. We start by evaluating `a` and finding value 2 in $E2$. Then, we evaluate `b` and cannot find it in $E2$...but we don't panic! We follow the parent pointer to $E1$ and try again. We find a binding for `b` in $E1$ and get the value 6. So, the value of `a + b` in $E2$ is 8.

Exercise 3.12
Draw a picture of the relevant binding environments when the statement with the arrow is being executed. What value does the procedure return?

```
def f(a):
    return a + g(a)

def g(b):
    return a + b    # <-------------
>>> f(3)
```

Exercise 3.13
Draw a picture of the relevant binding environments when the statement with the arrow is being executed. What value does the procedure return?

```
def f(a):
    def g(b):
        return a + b    # <-------------
    return a + g(a)

>>> f(3)
```

Exercise 3.14
Draw a picture of the relevant binding environments when the statement with the arrow is being executed. What value does the procedure return? Does it cause an error?

```
def a(a):
    return a * a  #<-------------

>>> a(2)
```

3.4.4 Environments in Python

Generally, Python establishes the following binding environments:

1 `__builtin__`: the mother of all environments: it contains the definitions of all sorts of basic symbols, like `list` and `sum`. It is the parent of all module environments.

2 Module: each separate file that contains Python code is called a module and establishes its own environment, whose parent is `__builtin__`.

3 Procedure calls: as described in this section. A procedure that is defined at the 'top level' of a module (that is, not nested in the definition of another procedure) has the module's environment as its parent, and has its name defined in the module's environment. Procedures that are defined inside other procedures have the procedure-call environment of the containing procedure as their parent.

We have seen two operations that cause bindings to be created: assignments and procedure calls. Bindings are also created when you evaluate an `import` statement. If you evaluate

```
import math
```

then a file associated with the `math` module is evaluated and the name `math` is bound, in the current environment, to the math module, which is an environment. No other names are added to the current environment, and if you want to refer to names in that module, you have to qualify them, as in `math.sqrt`. (As we will see in a subsequent section, the evaluation of this expression first evaluates `math`, which returns an environment. We can then evaluate `sqrt` with respect to that environment, thus returning a pointer to the procedure stored there.) If you execute

```
from math import sqrt
```

then the `math` file is evaluated, and the name `sqrt` is bound, in the current environment, to whatever the name `sqrt` is bound in the `math` module. But note that if you do this, the name `math` is not bound to anything, and you cannot access any other procedures in the `math` module unless you import them explicitly, as well.

3.4.5 Non-Local References in Procedures

There is an important subtlety in the way names are handled in the environment created by a procedure call. When a name that is not bound in the local environment is referenced, then it is looked up in the chain of parent environments. So, as we have seen, it is fine to have

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

When a name is assigned inside a procedure, a new binding is created for it in the environment associated with the current call of that procedure. So, it is fine to have

```
a = 2
def b():
    a = 3
    c = 4
    return 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 return expression. It will not change `a` in the global environment.

But here is a code fragment that causes trouble:

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

It seems completely reasonable, and you might expect `b()` to return 4. But, instead, it generates an error. What is 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` occurs on the left-hand-side of an assignment statement, and so, at the very beginning, it puts a new entry for `a` in the local environment, but without any value bound to it. Now, when it is time to evaluate the statement

```
a = a + 1
```

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

In Python, we can write code to increment 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
>>> a
5
```

The statement `global a` asks that a new binding for `a` *not* be made in the procedure-call environment. Now, all references to `a` are to the binding in the module's environment, and so this procedure actually changes `a`.

In Python, we can only make assignments to names in the procedure-call environment or to the module environment, but not to names in intermediate environments. 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 would really like `inner` to be able to see and modify the `a` that belongs to the environment for `outer`, but there is no way to arrange this. Some other programming languages, such as Scheme, offer more fine-grained control over how the scoping of variables is handled.

3.4.6 Procedures as first-class objects

In Python, unlike many other languages, procedures are treated in much the same way as numbers: they can be stored as values of variables, can be passed as arguments to procedures, and can be returned as results of procedure calls. Because of this, we say that procedures are treated as "first-class" objects. We will explore this treatment of "higher-order" procedures (procedures that manipulated procedures) throughout this section.

First of all, it is useful to see that we can construct (some) procedures without naming them using the `lambda` constructor:

```
lambda <var1>, ..., <varn> : <expr>
```

The formal parameters are `<var1>, ..., <varn>` and the body is `<expr>`. There is no need for an explicit `return` statement; 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 examples:

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

Here is a procedure of two arguments defined with `lambda`:

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

Using the expression-evaluation rules we have already established, we can do some fancy things with procedures, which we will illustrate throughout the rest of this section.

```
>>> procs = [lambda x: x, lambda x: x + 1, lambda x: x + 2]
>>> procs[0]
<function <lambda> at 0x83d70>
>>> procs[1](6)
7
```

Here, we have defined a list of three procedures. We can see that an individual element of the list (e.g., `procs[0]`) is a procedure.[8] So, then `procs[1](6)` applies the second procedure in the list to the argument 6. Since the second procedure returns its argument plus 1, the result is 7.

Here is a demonstration that procedures can be assigned to names in just the way other values can.

```
>>> fuzz = procs[2]
>>> fuzz(3)
5

def thing(a):
    return a * a

>>> thing(3)
9
>>> thang = thing
>>> thang(3)
9
```

Passing procedures as arguments

Just as we can pass numbers or lists into procedures as arguments, we can pass in procedures as arguments, as well.

What if we find that we are often wanting to perform the same procedure twice on an argument? That is, we seem to keep writing `square(square(x))`. If it were always the same procedure we were applying twice, we could just write a new procedure

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

But what if it is different procedures? We can write a new procedure that takes a procedure `f` as an argument and applies it twice:

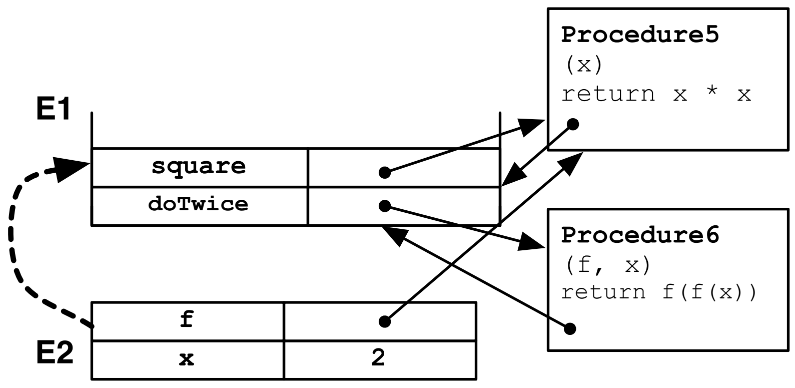
```
def doTwice(f, x):
    return f(f(x))
```

So, if we wanted to square twice, we could do:

```
>>> doTwice(square,2)
16
```

Here is a picture of the environment structure in place when we are evaluating the `return f(f(x))` statement in `doTwice`: Environment $E1$ is the module environment, where

[8]In the programming world, people often use the words "function" and "procedure" either interchangeable or with minor subtle distinctions. The Python interpreter refers to the objects we are calling procedures as "functions."



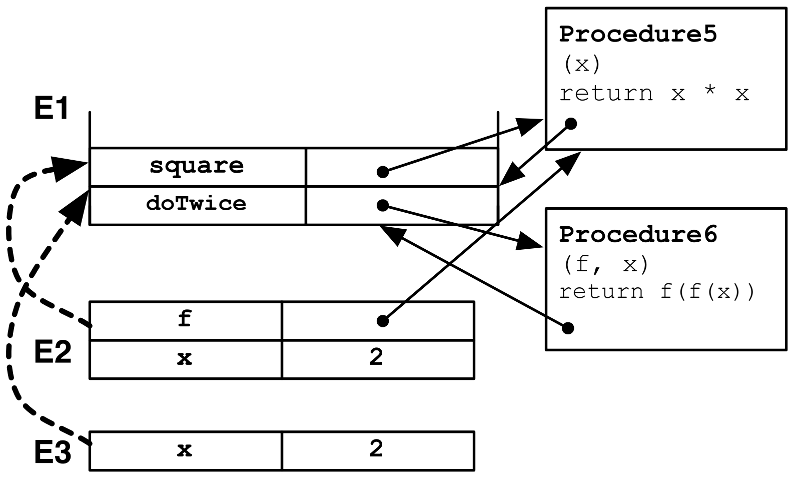
procedures `square` and `doTwice` were defined and where the expression `doTwice(square, 2)` is being evaluated. The interpreter:

- Evaluates `doTwice` in $E1$ and gets `Procedure6`.
- Evaluates `square` in $E1$ and gets `Procedure5`.
- Evaluates `2` in $E1$ and gets 2.
- Makes the new binding environment $E2$, binding the formal parameters, `f` and `x`, of `Procedure 6`, to actual arguments `Procedure5` and 2.
- Evaluates the body of `Procedure6` in $E2$.

Now, let's peek one level deeper into the process. To evaluate `f(f(x))` in $E2$, the interpreter starts by evaluating the inner `f(x)` expression. It

- Evaluates `f` in $E2$ and gets `Procedure5`.
- Evaluates `x` in $E2$ and gets 2.
- Makes the new binding environment $E3$, binding the formal parameter `x`, of `Procedure5`, to 2. (Remember that the parent of $E3$ is $E1$ because `Procedure5` has $E1$ as a parent.)
- Evaluates the body of `Procedure5` in $E3$, getting 4.

The environments at the time of this last evaluation step are:



A similar thing happens when we evaluate the outer application of `f`, but now with argument 4, and a return value of 16.

Exercise 3.15
Here is the definition of a procedure `sumOfProcs` that takes two procedures, `f` and `g`, as well as another value `x`, as arguments, and returns `f(x) + g(x)`. The `sumOfProcs` procedure is then applied to two little test procedures:

```
def sumOfProcs(f, g, x):
    return f(x) + g(x)

def thing1(a):
    return a*a*a
def thing2(b):
    return b+1    #   <--------------------

>>> sumOfProcs(thing1, thing2, 2)
```

Draw a picture of all of the relevant environments at the moment the statement with the arrow is being evaluated. What is the return value of the call to `sumOfProcs`?

Returning procedures as values

Another way to apply a procedure multiple times is this:

```
def doTwiceMaker(f):
    return lambda x: f(f(x))
```

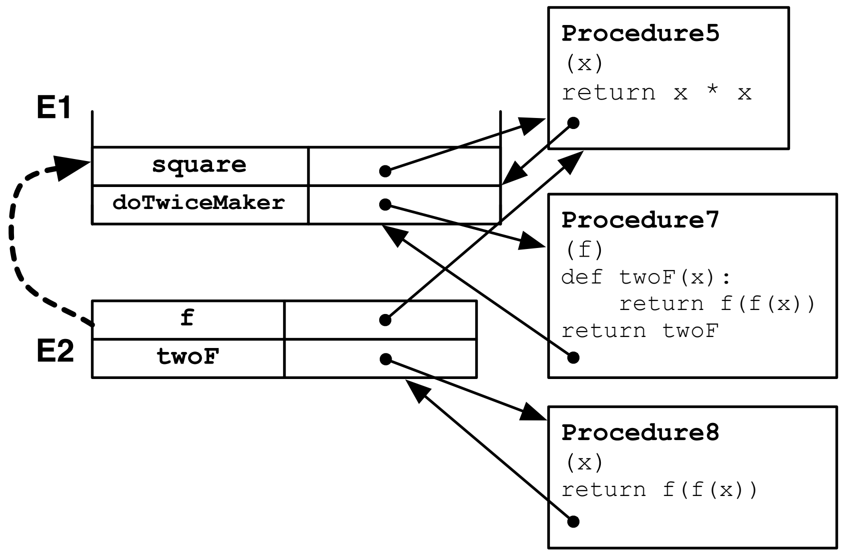
This is a procedure that *returns a procedure*! If you would rather not use `lambda`, you could write it this way:

```
def doTwiceMaker(f):
    def twoF(x):
        return f(f(x))
    return twoF
```

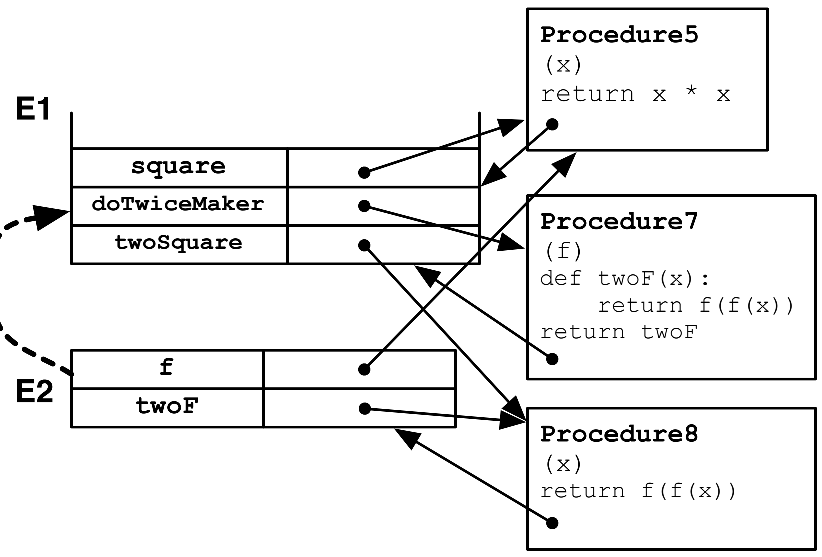
Now, to use `doTwiceMaker`, we might start by calling it with a procedure, such as `square`, as an argument and naming the resulting procedure.

```
>>> twoSquare = doTwiceMaker(square)
```

Here is a picture of the environments just before the `return twoF` statement in `doTwiceMaker` is evaluated.



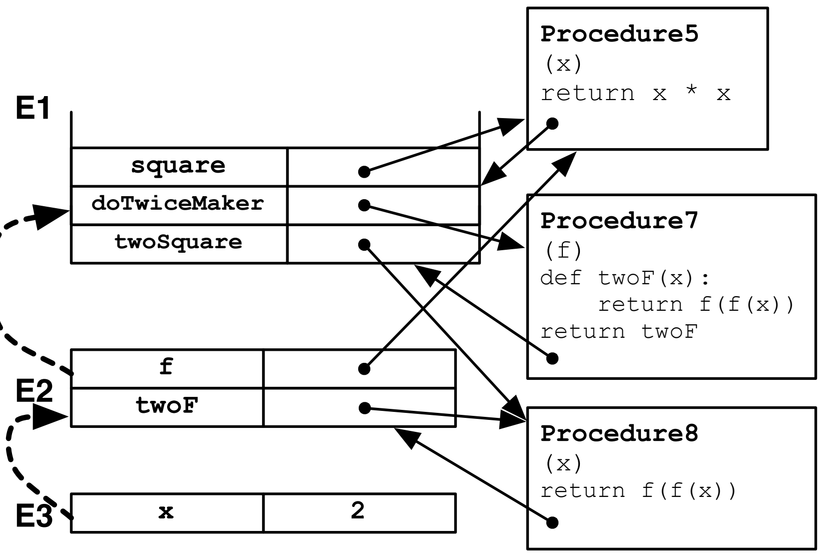
Here is a picture of the environments after the `doTwiceMaker` returns its value and it is assigned to `twoSquare` in $E1$. It is important to see that `Procedure8` is the return value of the call to `doTwiceMaker` and that, because `Procedure8` retains a pointer to the environment in which it was defined, we need to keep $E2$ around. And it is $E2$ that remembers which procedure (via its binding for `f`) is going to be applied twice.



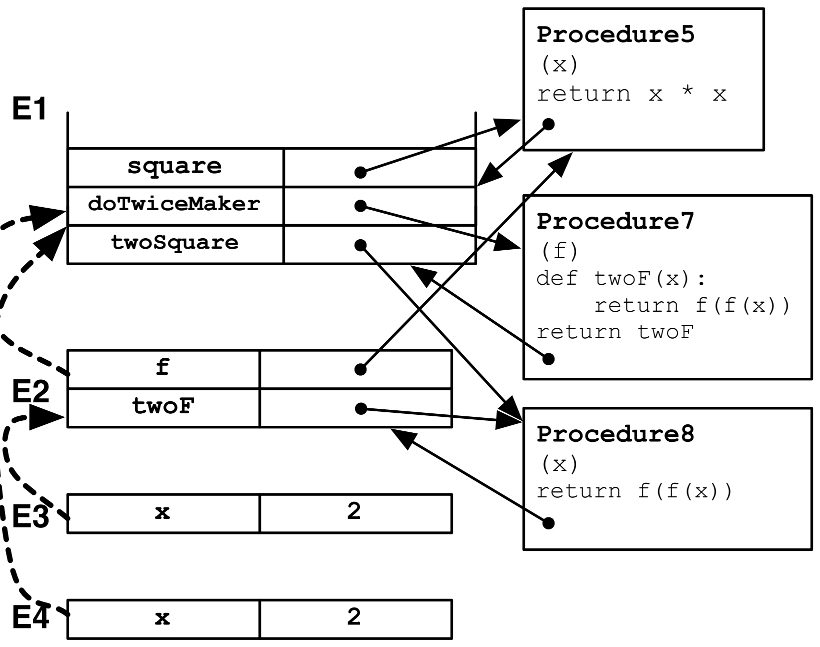
Now, when we evaluate this expression in $E1$

```
>>> twoSquare(2)
```

we start by making a new binding environment, $E3$, for the procedure call. Note that, because the procedure we are calling, `Procedure8`, has $E2$ stored in it, we set the parent of $E3$ to be $E2$.



Next, we evaluate the body of `Procedure8`, which is `return f(f(x))` in $E3$. Let's just consider evaluating the inner expression `f(x)` in $E3$. We evaluate `f` in $E3$ and get `Procedure5`, and evaluate `x` and get 2. Now, we make a new binding environment, $E4$, to bind the formal parameter of `Procedure5` to 2. Because `Procedure5` has a stored pointer to $E1$, $E4$'s parent is $E1$, as shown here:



Evaluating the body of `Procedure5` in $E4$ yields 4. We will repeat this process to evaluate the outer application of `f`, in `f(f(x))`, now with argument 4, and end with result 16.

Essentially the same process would happen when we evaluate

```
>>> doTwiceMaker(square)(2)
16
```

except the procedure that is created by the expression `doTwiceMaker(square)` is not assigned a name; it is simply used as an intermediate result in the expression evaluation.

3.5 Object-Oriented Programming

We have seen structured data and interesting procedures that can operate on that data. It will often be useful to make a close association between collections of data and the operations that apply to them. The style of programming that takes this point of view is *object-oriented programming* (OOP). It requires adding some simple mechanisms to our interpreter, but is not a big conceptual departure from the things we have already seen. It is, however, a different style of organizing large programs.

3.5.1 Classes and Instances

In OOP, we introduce the idea of *classes* and *instances*. An *instance* is a collection of data that describe a single entity in our domain of interest, such as a person or a car or a point in 3D space. If we have many instances that share some data values, or upon which we would want to perform similar operations, then we can represent them as being members of a *class* and store the shared information once with the class, rather than replicating it in the instances.

Consider the following staff database for a large undergraduate course:

name	role	age	building	room	course
Pat	Prof	60	34	501	6.01
Kelly	TA	31	34	501	6.01
Lynn	TA	29	34	501	6.01
Dana	LA	19	34	501	6.01
Chris	LA	20	34	501	6.01

There are lots of shared values here, so we might want to define a class. A class definition has the form:

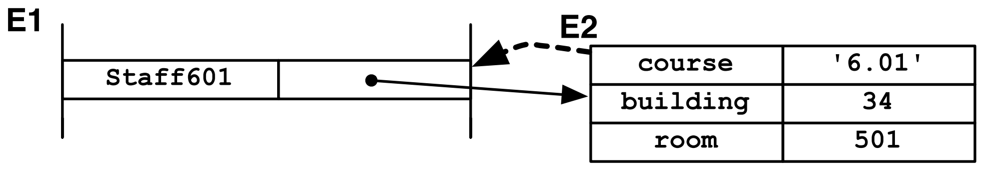
```
class <name>:
    <statement1>
    ...
    <statementn>
```

Here is the definition of simple class in our example domain:

```
class Staff601:
    course = '6.01'
    building = 34
    room = 501
```

From the implementation perspective, the most important thing to know is that **classes and instances are environments**.

When we define a new class, we make a new environment. In this case, the act of defining `class Staff601` in an environment $E1$ results in a binding from `Staff601` to $E2$, an empty environment whose parent is $E1$, the environment in which the class definition was evaluated. Now, the statements inside the class definition are evaluated in the new environment, resulting in a memory state like this:[9] Note how the common values and



names have been captured in a separate environment.

[9]In fact, the string `'6.01'` should be shown as an external memory structure, with a pointer stored in the binding environment; for compactness in the following diagrams we will sometimes show the strings themselves as if they were stored directly in the environment.

Caveat: when we discuss methods in Section 3.5.2, we will see that the rules for evaluating procedure definitions inside a class definition are slightly different from those for evaluating procedure definitions that are not embedded in a class definition.

We will often call names that are bound in a class's environment *attributes* of the class. We can access these attributes of the class after we have defined them, using the *dot* notation:

```
<envExpr>.<var>
```

When the interpreter evaluates such an expression, it first evaluates `<envExpr>`; if the result is not an environment, then an error is signaled. If it is an environment, E, then the name `<var>` is looked up in E (using the general process of looking in the parent environment if it is not found directly in E) and the associated value returned.

So, for example, we could do:

```
>>> Staff601.room
501
```

We can also use the dot notation on the left-hand side of an assignment statement, if we wish to modify an environment. An assignment statement of the form

```
<envExpr>.<var> = <valExpr>
```

causes the name `<var>` in the environment that is the result of evaluating `<envExpr>` to be bound to the result of evaluating `<valExpr>`.

So, we might change the room in which 6.01 meets with:

```
Staff601.room = Staff601.room - 100
```

or add a new attribute with

```
Staff601.coolness = 11  # out of 10, of course...
```

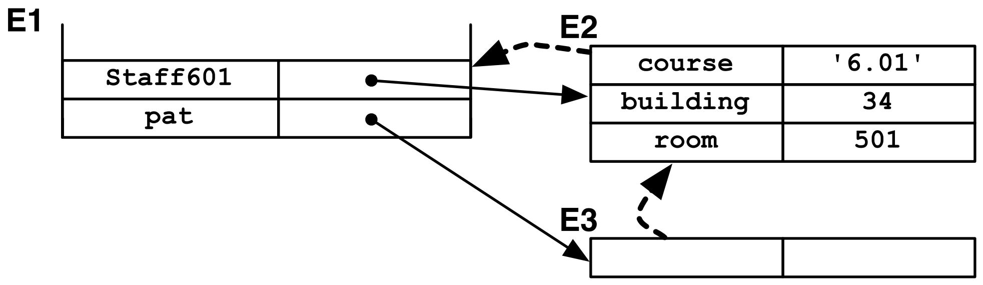
Now, we can make an *instance* of a class with an expression of the form:

```
<classExpr>()
```

This expression has as its value a new empty environment whose parent pointer is the environment obtained as a result of evaluating `<classExpr>`. [10]

So, if we do:

```
>>> pat = Staff601()
```



we will end up with an environment state like this:

At this point, given our standard rules of evaluation and the dot notation, we can say:

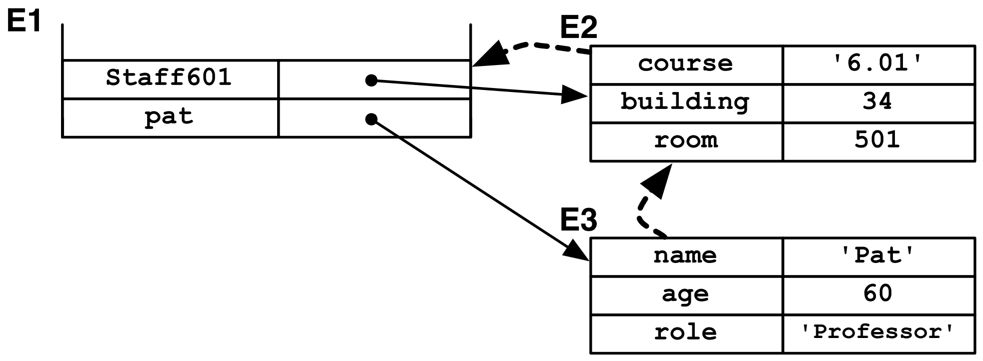
```
>>> pat.course
'6.01'
```

The interpreter evaluates `pat` in $E1$ to get the environment $E3$ and then looks up the name `course`. It does not find it in $E3$, so it follows the parent pointer to $E2$, and finds there that it is bound to `'6.01'`.

Similarly, we can set attribute values in the instance. So, if we were to do:

```
pat.name = 'Pat'
pat.age = 60
pat.role = 'Professor'
```

we would get this environment structure.



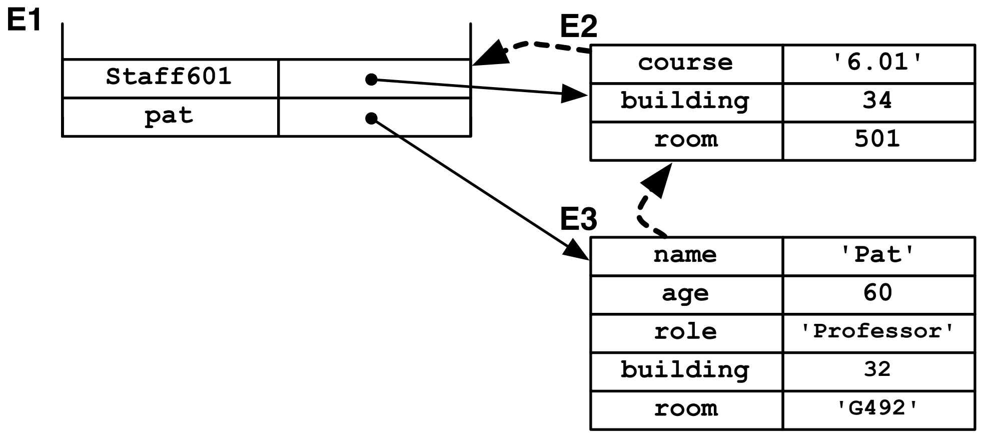
Note that these names are bound in the instance environment, not the class.

These structures are quite flexible. If we wanted to say that Professor Pat is an exception, holding office hours in a different place from the rest of the 6.01 staff, we could say:

```
pat.building = 32
pat.office = 'G492'
```

[10]Another way of thinking of this is that whenever you define a class, Python defines a procedure with the same name, which is used to create a instance of that class.

Here is the new environment state. Nothing is changed in the `Staff601` class: these assignments just make new bindings in `pat`'s environment, which 'shadow' the bindings in the class, so that when we ask for `pat`'s building, we get 32.



3.5.2 Methods

Objects and classes are a good way to organize procedures, as well as data. When we define a procedure that is associated with a particular class, we call it a *method* of that class. Method definition requires only a small variation on our existing evaluation rules.

So, imagine we want to be able to greet 6.01 staff members appropriately on the staff web site. We might add the definition of a `salutation` method:

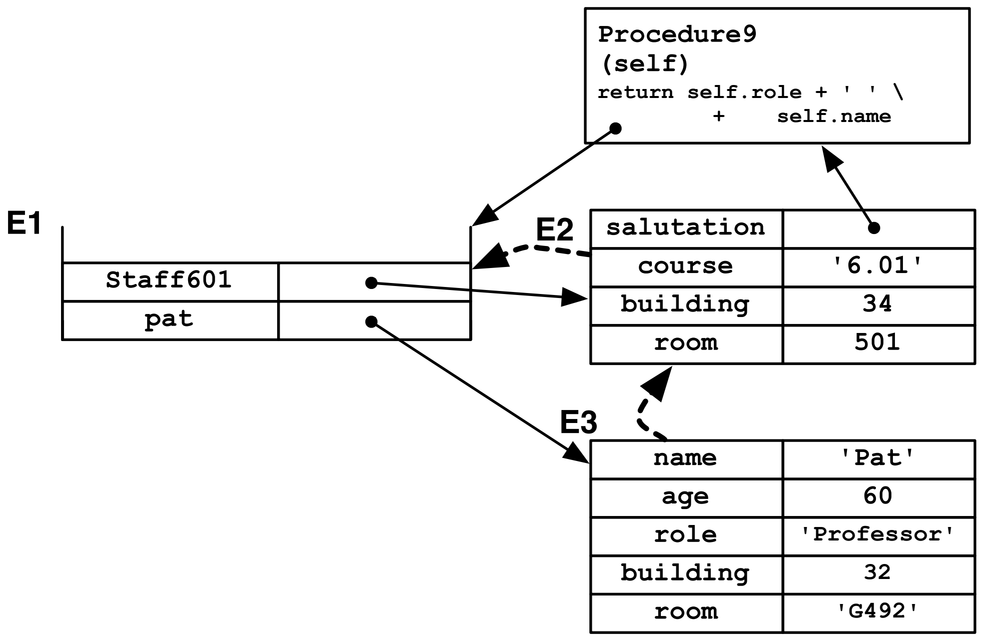
```
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def salutation(self):
        return self.role + ' ' + self.name
```

This procedure definition, made inside the class definition, is evaluated in *almost* the standard way, resulting in a binding of the name `salutation` in the class environment to the new procedure. The way in which this process deviates from the standard procedure evaluation process is that the environment stored in the procedure is the module (file) environment, no matter how deeply nested the class and method definitions are:

Now, for example, we could do:

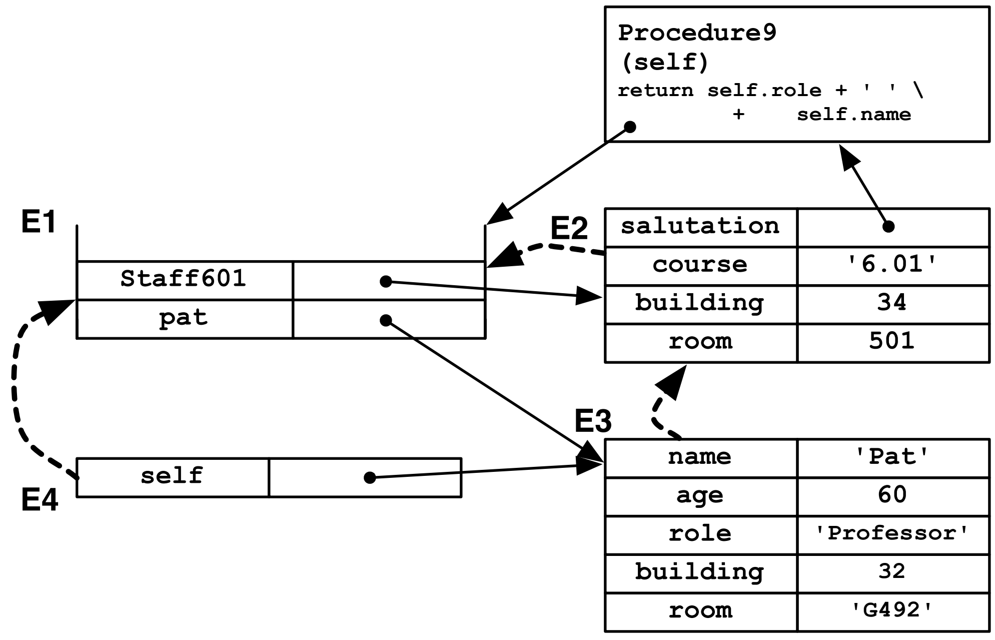
```
Staff601.saluation(pat)
```

The interpreter finds that `Staff601` is an environment, in which it looks up the name `saluation` and finds `Procedure9`. To call that procedure we follow the same steps as in Section 3.4.2 :



- Evaluate `pat` to get the instance $E3$.
- Make a new environment, $E4$, binding `self` to $E3$. The parent of $E4$ is $E1$, because we are evaluating this procedure call in $E1$.
- Evaluate `self.role + ’ ’ + self.name` in $E4$.
- In $E4$, we look up `self` and get $E3$, look up `role` in $E3$ and get `’Professor’`, etc.
- Ultimately, we return `’Professor Pat’`.

Here is a picture of the binding environments while we are evaluating `self.role + ’ ’ + self.name`.



Note (especially for Java programmers!) that the way the body of `salutation` has access to the attributes of the instance on which it is operating and to attributes of the class is via the instance we passed to it as the parameter `self`. The parent environment is $E1$, which means that methods cannot simply use the names of class attributes without accessing them through the instance.

The notation

```
Staff601.salutation(pat)
```

is a little clumsy; it requires that we remember to what class `pat` belongs, in order to get the appropriate `salutation` method. We ought, instead, to be able to write

```
pat.salutation(pat)     #### Danger:  not legal Python
```

This should have exactly the same result. (Verify this for yourself if you do not see it, by tracing through the environment diagrams. Even though the `pat` object has no binding for `saluation`, the environment-lookup process will proceed to its parent environment and find a binding in the class environment.)

But this is a bit redundant, having to mention `pat` twice. So, Python added a special rule that says: *If you access a class method by looking in an* **instance***, then that instance is automatically passed in as the first argument of the method.*

So, we can write

```
pat.salutation()
```

This is **exactly** equivalent to

```
Staff601.salutation(pat)
```

A consequence of this decision is that every method must have an initial argument that is an instance of the class to which the method belongs. That initial argument is traditionally called `self`, but it is not necessary to do so.

Here is a new method. Imagine that `Staff601` instances have a numeric `salary` attribute. So, we might say

```
pat.salary = 100000
```

Now, we want to write a method that will give a 6.01 staff member a k-percent raise:

```
class Staff601:
    course = '6.01'
    building = 34
    room = 501
    def salutation(self):
        return self.role + ' ' + self.name

    def giveRaise(self, percentage):
        self.salary = self.salary + self.salary * percentage
```

As before, we could call this method as

```
Staff601.giveRaise(pat, 0.5)
```

or we could use the short-cut notation and write:

```
pat.giveRaise(0.5)
```

This will change the `salary` attribute of the `pat` instance to 150000.[11]

3.5.3 Initialization

When we made the `pat` instance, we first made an empty instance, and then added attribute values to it. Repeating this process for every new instance can get tedious, and we might wish to guarantee that every new instance we create has some set of attributes defined. Python has a mechanism to streamline the process of initializing new instances. If we define a class method with the special name `__init__`, Python promises to call that method whenever a new instance of that class is created.

We might add an initialization method to our `Staff601` class:

```
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def __init__(self, name, role, years, salary):
        self.name = name
        self.role = role
        self.age = years
        self.salary = salary

    def salutation(self):
```

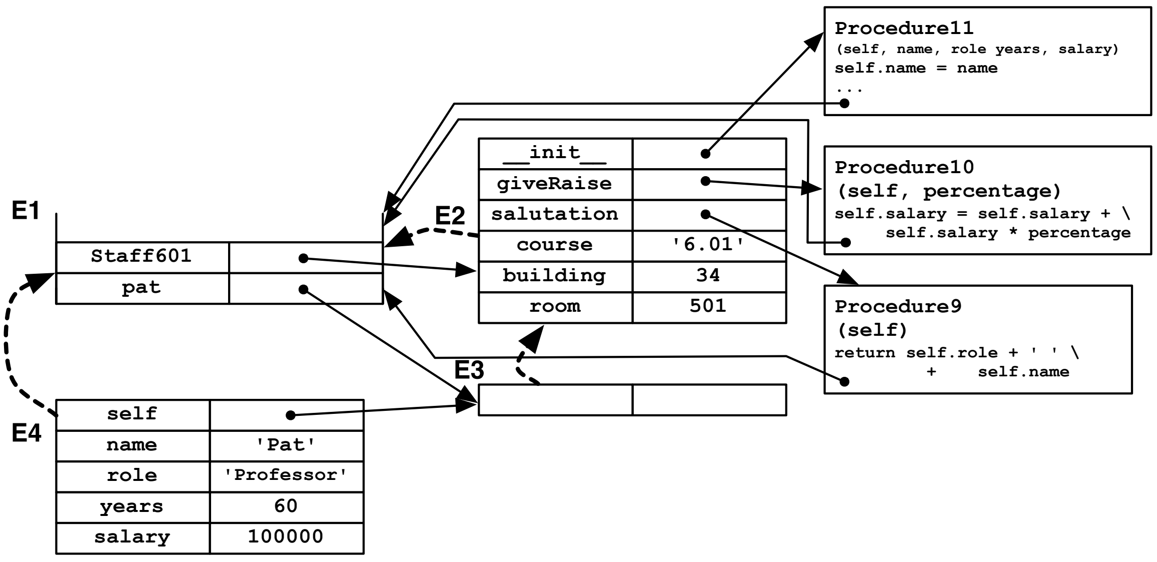
[11]Something to watch out for!!! A common debugging error happens when you make an instance of a class (such as our `pat` above); then change the class definition and re-evaluate the file; then try to test your changes using your old instance, `pat`. Instances remember the definitions of all the methods in their class *when they were created*. So if you change the class definition, you need to make a new instance (it could still be called `pat`) in order to get the new definitions.

```
        return self.role + ' ' + self.name

    def giveRaise(self, percentage):
        self.salary = self.salary + self.salary * percentage
```

Now, to create an instance, we would do:[12]

```
pat = Staff601('Pat', 'Professor', 60, 100000)
```

Here is a diagram of the environments when the body of the `__init__` procedure is about to be executed: Note that the formal parameter `self` has been bound to the newly-

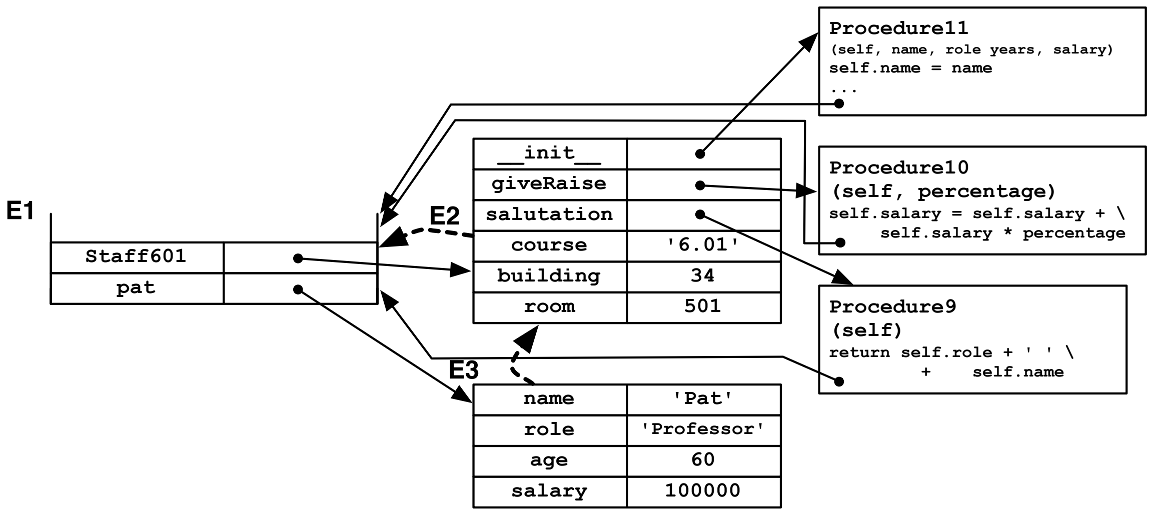


created instance. The diagram below shows the situation after the initialization method has finished executing.

This method seems very formulaic, but it is frequently all we need to do. To see how initialization methods may vary, we might instead do something like this, which sets the salary attribute based on the role and age arguments passed into the initializer.

```
class Staff601:
    def __init__(self, name, role, age):
        self.name = name
        self.role = role
        if self.role == 'Professor':
            self.salary = 100000
        elif self.role = 'TA':
            self.salary = 30000
        else:
```

[12]We called the fourth formal parameter `years`, when `age` would have been clearer, just to illustrate that the names of formal parameters do not have to match the attributes to which they are bound inside the object.



```
            self.salary = 10000
        self.salary = self.giveRaise((age - 18) * 0.03)
```

3.5.4 Inheritance

We see that we are differentiating among different groups of 6.01 staff members. We can gain clarity in our code by building that differentiation into our object-oriented representation using *subclasses* and *inheritance*.

At the mechanism level, the notion of a subclass is very simple: if we define a class in the following way:

```
def class <className>(<superclassName):
    <body>
```

then, when the interpreter makes the environment for this new class, it sets the parent pointer of the class environment to be the environment named by `<superclassName>`.

This mechanism allows us to factor class definitions into related, interdependent aspects. For example, in the 6.01 staff case, we might have a base class where aspects that are common to all kinds of staff are stored, and then subclasses for different roles, such as professors:

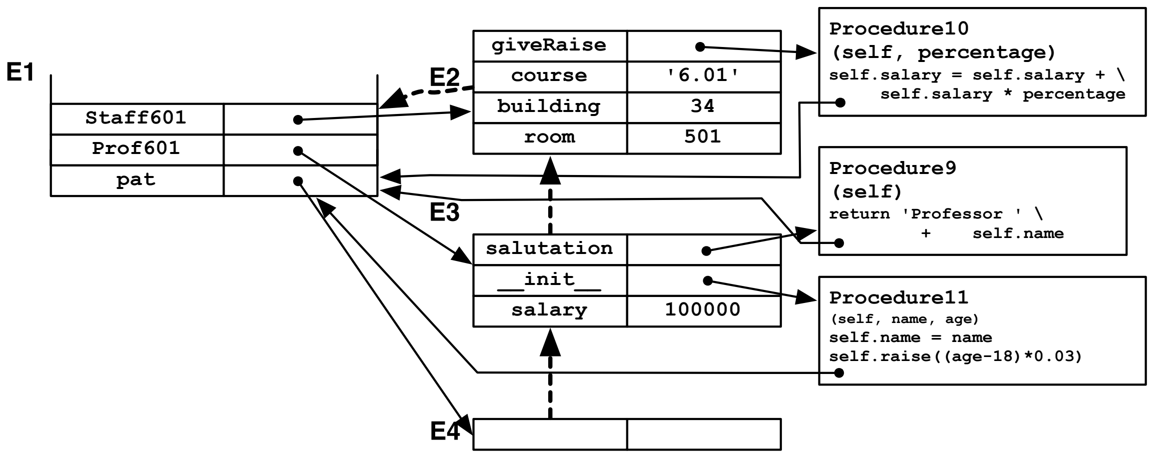
```
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def giveRaise(self, percentage):
        self.salary = self.salary + self.salary * percentage
```

```
class Prof601(Staff601):
    salary = 100000

    def __init__(self, name, age):
        self.name = name
        self.giveRaise((age - 18) * 0.03)

    def salutation(self):
        return 'Professor' + self.name
```

Let's trace what happens when we make a new instance of `Prof601` with the expression

```
Prof601('Pat', 60)
```

First a new environment, $E4$, is constructed, with $E3$ (the environment associated with the class `Prof601` as its parent). Here is the memory picture now:



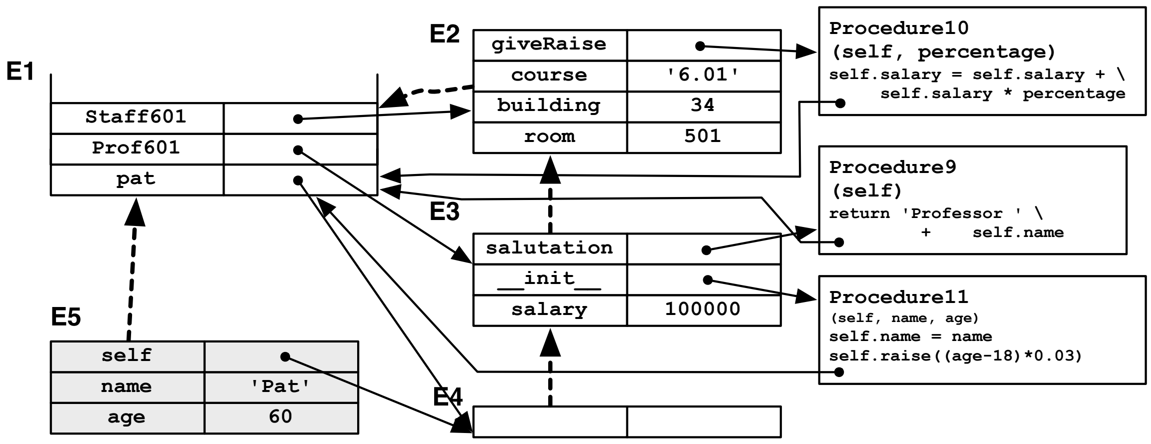
As soon as it is created, the `__init__` method is called, with the new environment $E4$ as its first parameter and the rest of its parameters obtained by evaluating the expressions that were passed into to `Prof601`, in this case, `'Pat'` and 60. Now, we need to make the procedure-call environment, binding the formal parameters of `Procedure11`; it is $E5$ in this figure: We evaluate the body of `Procedure11` in $E5$. It starts straightforwardly by evaluating

```
        self.name = name
```

which creates a binding from `name` to `'Pat'` in the object named `self`, which is $E4$. Now, it evaluates

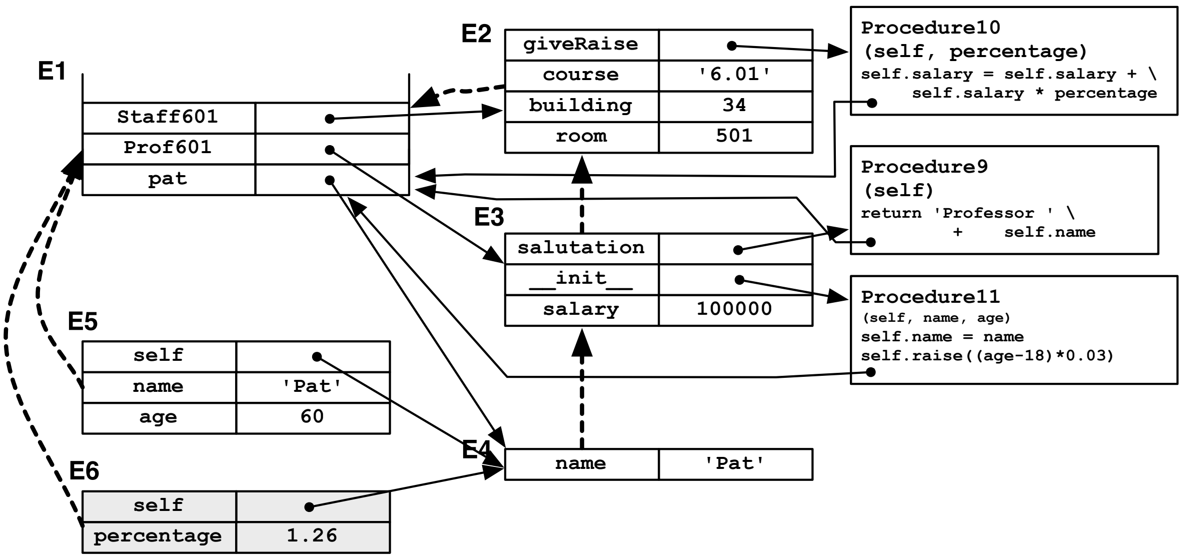
```
self.giveRaise((age - 18) * 0.03)
```

in $E5$. It starts by evaluating `self.giveRaise`. `self` is $E4$, so we look for a binding of `giveRaise`. It is not bound in $E4$, so we look in the parent $E3$; it is not bound in $E3$



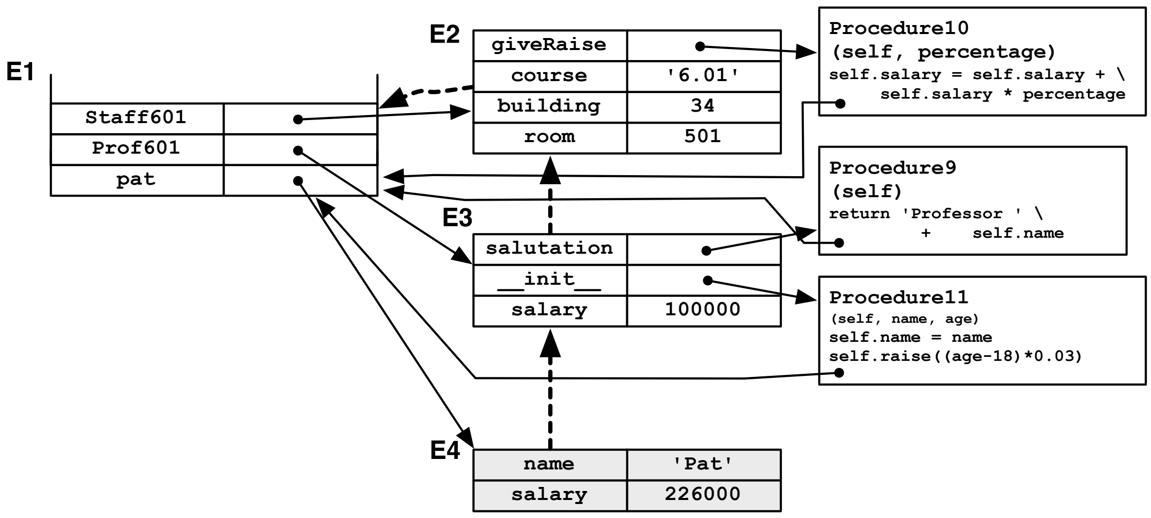
so we look in $E2$ and find that it is bound to `Procedure10`. We are taking advantage of the fact that raises are not handled specially for this individual or for the subclass of 6.01 professors, and use the definition in the general class of 6.01 staff. The interpreter evaluates the argument to `Procedure10`, `(60 - 18) * 0.03`, getting 1.26.

It is time, now, to call `Procedure10`. We have to remember that the first argument will be the object through which the method was accessed: $E4$. So, we make a binding environment for the parameters of `Procedure10`, called $E6$: Now the fun really starts! We



evaluate `self.salary = self.salary + self.salary * percentage` in $E6$. We start by evaluating the right hand side: `self` is $E4$, `self.salary` is 100000, and `percentage` is 1.26, so the right-hand side is 226000. Now, we assign `self.salary`, which means we make a binding in $E4$ for `salary`, to 22600. It is important to see that, within a method call, all access to attributes of the object, class, or superclass goes through `self`. It is not possible to 'see' the definition of the `building` attribute directly from the `giveRaise` method: if `giveRaise` needed to depend on the `building` attribute, it would need to access it via the object, with `self.building`. This guarantees that we always get the definition of any attribute or method that is appropriate for that particular object, which may have overridden its definition in the class.

When the procedure calls are all done, the environments are finally like this: It is useful



to back up and see the structure here.

- The instance `pat` is an environment, $E4$.
- The parent of the instance is an environment,$E3$, which is the class `Prof601`.
- The parent of the class is an environment, $E2$, which the superclass `Staff601`.

3.5.5 Using Inheritance

Frequently when we're using inheritance, we override some methods of a parent class, and use some of its methods unchanged. Occasionally, we will want to modify or augment an existing method from the parent class. In such a situation, we can 'wrap' that method by writing our own version of that method in the child class, but calling the method of the parent to do some of the work. To call a method `bar` from a parent class named `Foo`, you can do something like the following.

```
def SubclassOfFoo(Foo):
    def bar(self, arg):
        Foo.bar(self, arg)
```

When you call a method using the class name, rather than an object name, you need to pass the object in explicitly to the method call.

Here is a concrete example. Imagine that you have a class for managing calendars. It has these methods, plus possibly some others.

```
class Calendar:
    def __init__(self):
        self.appointments = []
    def makeAppointment(self, name, dateTime):
        self.appointments.append((name, dateTime))
    def printCalendar(self, start, end):
        # complicated stuff
```

Now, you'd like to make a calendar that can handle appointments that are made in different time zones. It should store all of the appointments in a single time zone, so that it can sort and display appropriately.

```
class CalendarTimeZone(Calendar):
    def __init__(self, zone):
        Calendar.__init__(self)
        self.zone = zone

    def makeAppointment(self, name, dateTime):
        Calendar.makeAppointment(self, name, dateTime + zone)
```

We make a subclass of `Calendar` which takes a time zone `zone` as an argument to its initialization method. It starts by calling the `Calendar` initialization method, to set up whatever internal data structures are necessary, and then sets the `zone` attribute of the new object. Now, when it's time to make an appointment, we add the time zone offset to the appointed time, and then call the `makeAppointment` method of `Calendar`.

What is nice about this example is that we were able to take advantage of the `Calendar` class, and even add to some of its methods, without knowing anything about its internal representation of calendars.

3.6 Recursion

There are many control structures in Python, and other modern languages, which allow you write short programs that do a lot of work. In this section, we discuss recursion, which is also a way to write programs of arbitrary complexity. It is of particular importance here, because the structure of a language interpreter is recursive, and in the next section we will explore, in detail, how to construct an interpreter.

We have seen how we can define a procedure, and then 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 are in the middle of defining as a black box. That is 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 is 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 is sure to happen. In the base case, you just compute the answer directly (no more calls to the recursive procedure!) and return it. Otherwise, you are 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 procedure, 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.

Another way to think about it is this: when you are given a problem of size n, assume someone already gave you a way to solve problems of size $n - 1$. So, now, your job is only to figure out

- To which problem of size $n - 1$ you would like to know the answer, and
- How to do some simple operations to make that answer into the answer to your problem of size n.

What if you wanted to add two positive numbers, but your computer only had the ability to increment and decrement (add and subtract 1)? You could do this recursively, by thinking about it like this. I need to add `m` and `n`:

- Presupposing the ability to solve simpler problems, I could get the answer to the problem `m` plus `n-1`.
- Now, I just need to add 1 to that answer, to get the answer to my problem.

We further need to reason that when `n` is 0, then the answer is just `m`. This leads to the following Python definition:

```
def slowAdd(m, n):
    if n == 0:
        return m
    else:
        return 1 + slowAdd(m, n-1)
```

Note how in the final `return` expression, we have reduced the answer to a problem of size `n` to a simpler version of the same problem (of size `n-1` plus some simple operations.

Here is 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 is 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.

Exercise 3.16
What is the result of evaluating `bunchaOnes(-5)`?

Here is another example. It is 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, 4)`, by adding a print statement that prints arguments `a` and `b` as the first line of the procedure, and seeing what happens.

Here is 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 are down to a 1 or a 0, in which case the answer is obvious. If we do not have an easy case, we divide up our problem into two that are easier. So, if we convert `n/2` (the integer result of dividing `n` by 2, which by Python's definition will be a smaller number since it truncates the result, throwing away any remainder), into a string of digits, we will have all but the last digit. And `n%2` (`n` modulo 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 is 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 on which it is operating is a positive integer that 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 a way to add up the values of a list of numbers:

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

The `addList` procedure consumed a list and produced a number. The `incrementElements` 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 incrementElements(elts):
    if elts == []:
        return []
    else:
        return [elts[0]+1] + incrementElements(elts[1:])
```

If the list of elements is empty, then there is no work to be done, and the result is just the empty list. Otherwise, the result is a new list: the first element of the new list is the first element of the old list, plus 1; the rest of the new list is the result of calling `incrementElement` recursively on the rest of the input list. Because the list we are operating on is getting shorter on every recursive call, we know we will reach the base case, and all will be well.

3.7 Object-Oriented Programming Examples

Here are some examples of object-oriented programming in Python.

3.7.1 A simple method example

Here's an example to illustrate the definition and use of classes, instances, and methods.

```
class Square:
    def __init__(self, initialDim):
        self.dim = initialDim
    def getArea (self):
        return self.dim * self.dim

    def setArea (self, area):
        self.dim = area**0.5

    def __str__(self):
        return "Square of dim " + str(self.dim)
```

This class is meant to represent a square. Squares need to store, or remember, their dimension, so we make an attribute for it, and assign it in the `__init__` method. Now, we define a method `getArea` that is intended to return the area of the square. There are a couple of interesting things going on here.

Like all methods, `getArea` has an argument, `self`, which will stand for the instance that this method is supposed to operate on. Now, the way we can find the dimension of the square is by finding the value of the `dim` attribute of `self`.

We define another method, `setArea`, which will set the area of the square to a given value. In order to change the square's area, we have to compute a new dimension and store it in the `dim` attribute of the square instance.[13]

Now, we can experiment with instances of class `Square`.

```
>>> s = Square(6)
>>> s.getArea()
36
>>> Square.getArea(s)
36
>>> s.dim
6
>>> s.setArea(100)
>>> s.dim
10.0
>>> s1 = Square(10)
>>> s1.dim
10
>>> s1.getArea()
100
>>> s2 = Square(100)
>>> s2.getArea()
10000
>>> print s1
Square of dim 10
```

Our class `Square` has the `__str__` method defined, so it prints out nicely.

[13]We compute the square root of a value by raising to the power 0.5.

3.7.2 Superclasses and inheritance

What if we wanted to make a set of classes and instances that would help us to run a bank? We could start like this:

```
class Account:
    def __init__(self, initialBalance):
        self.currentBalance = initialBalance
    def balance(self):
        return self.currentBalance
    def deposit(self, amount):
        self.currentBalance = self.currentBalance + amount
    def creditLimit(self):
        return min(self.currentBalance * 0.5, 10000000)

>>> a = Account(100)
>>> b = Account(1000000)

>>> Account.balance(a)
100
>>> a.balance()
100
>>> Account.deposit(a, 100)
>>> a.deposit(100)
>>> a.balance()
300
>>> b.balance()
1000000
```

We've made an `Account` class[14] that maintains a balance as an attribute. There are methods for returning the balance, for making a deposit, and for returning the credit limit.

The `Account` class contains the procedures that are common to all bank accounts; the individual instances contain state in the values associated with the names in their environments. That state is *persistent*, in the sense that it exists for the lifetime of the program that is running, and doesn't disappear when a particular method call is over.

Now, imagine that the bank we're running is getting bigger, and we want to have several different kinds of accounts, with the credit limit depending on the account type. If we wanted to define another type of account as a Python class, we could do it this way:

[14]A note on style. It is useful to adopt some conventions for naming things, just to help your programs be more readable. We've used the convention that variables and procedure names start with lower case letters and that class names start with upper case letters. And we try to be consistent about using something called "camel caps" for writing compound words, which is to write a compound name with the successiveWordsCapitalized. An alternative is_to_use_underscores.

```
class PremierAccount:
    def __init__(self, initialBalance):
        self.currentBalance = initialBalance
    def balance(self):
        return self.currentBalance
    def deposit(self, amount):
        self.currentBalance = self.currentBalance + amount
    def creditLimit(self):
        return min(self.currentBalance * 1.5, 10000000)
>>> c = PremierAccount(1000)
>>> c.creditLimit()
1500.0
```

This will let people with premier accounts have larger credit limits. And, the nice thing is that we can ask for its credit limit without knowing what kind of an account it is, so we see that objects support *generic functions*, which operate on different kinds of objects by doing different, but type-appropriate operations.

However, this solution is still not satisfactory. In order to make a premier account, we had to repeat a lot of the definitions from the basic account class, violating our fundamental principle of laziness: never do twice what you could do once; instead, abstract and reuse.

Inheritance lets us make a new class that's like an old class, but with some parts overridden or new parts added. When defining a class, you can actually specify an argument, which is another class. You are saying that this new class should be exactly like the *parent class* or *superclass*, but with certain definitions added or overridden. So, for example, we can say

```
class PremierAccount(Account):
    def creditLimit(self):
        return min(self.currentBalance * 1.5, 10000000)

class EconomyAccount(Account):
    def creditLimit(self):
        return min(self.currentBalance*0.5, 20.00)

>>> a = Account(100)
>>> b = PremierAccount(100)
>>> c = EconomyAccount(100)
>>> a.creditLimit()
100.0
>>> b.creditLimit()
150.0
>>> c.creditLimit()
20.0
```

We automatically inherit the methods of our superclass (including `__init__`). So we still know how to make deposits into a premier account:

```
>>> b.deposit(100)
>>> b.balance()
200
```

The fact that instances know what class they were derived from allows us to ask each instance to do the operations appropriate to it, without having to take specific notice of what class it is. Procedures that can operate on instances of different types or classes without explicitly taking their types or classes into account are called *polymorphic*. Polymorphism is a very powerful method for capturing common patterns.

3.7.3 A data type for times

Object-oriented programming is particularly useful when we want to define a new compositional data type: that is, a representation for the data defining a class of objects and one or more operations for creating new instances of the class from old instances. We might do this with complex numbers or points in a high-dimensional space.

Here we demonstrate the basic ideas with a very simple class for representing times.

We'll start by defining an initialization method, with default values specified for the parameters.

```
class Time:
    def __init__(self, hours = 0, minutes = 0):
        self.hours = hours
        self. minutes = minutes
```

Here is a method for adding two time objects. If the summed minutes are greater than 60, it carries into the hours. If the summed hours are greater than 24, then the excess is simply thrown away. It is important that adding two `Times` returns a new `Time` object, and that it *does not change any aspect of either of the arguments.*

```
    def add(self, other):
        newMinutes = (self.minutes + other.minutes) % 60
        carryHours = (self.minutes + other.minutes) / 60
        newHours = (self.hours + other.hours + carryHours) % 24
        return Time(newHours, newMinutes)
```

Python has a special facility for making user-defined types especially cool: if, for instance, you define a method called __add__ for your class `Foo`, then whenever it finds an expression of the form `<obj1> + <obj2>`, and the expression `<obj1>` evaluates to an object of class `Foo`, it will `Foo`'s __add__ method. So, here we set that up for times:

```
    def __add__(self, other):
        return self.add(other)
```

Additionally, defining methods called `__str__` and `__repr__` that convert elements of your class into strings will mean that they can be printed out nicely by the Python shell.

```
    def __str__(self):
        return str(self.hours) + ':' + str(self.minutes)
    def __repr__(self):
        return str(self)
```

Here, we make some times. In the second case, we specify only the minutes, and so the hours default to 0.

```
>>> t1 = Time(6, 30)
>>> t2 = Time(minutes = 45)
>>> t3 = Time(23, 59)
```

And now, we can do some operations on them. Note how our `__str__` method allows them to be printed nicely.

```
>>> t1
6:30
>>> t1 + t2
7:15
>>> t3 + t2
0:44
>>> t1 + t2 + t3
7:14
>>> (t1 + t2 + t3).minutes
14
```

3.7.4 Practice problem: argmax

Write a Python procedure `argmax` that takes two arguments: the first is a list of elements, and the second is a function from elements to numerical values. It should return the element of the list with the highest numerical score.

Ans:

```
def argmax(elements, f):
    bestScore = None
    bestElement = None
    for e in elements:
        score = f(e)
        if bestScore == None or score > bestScore:
            bestScore = score
            bestElement = e
    return bestElement

def argmax(elements, f):
    bestElement = elements[0]
    for e in elements:
        if f(e) > f(bestElement):
           bestElement = e
    return bestElement

def argmax(elements, f):
    vals = [f(e) for e in elements]
    return elements[vals.index(max(vals))]

def argmax(elements, f):
    return max(elements, key=f)
```

There are many ways of writing this. Here are a few. It's important to keep straight in your head which variables contain scores and which contain elements. Any of these solutions would have been fine; we wouldn't expect most people to get the last one.

3.7.5 Practice problem: OOP

The following definitions have been entered into a Python shell:

```
class A:
    yours = 0
    def __init__(self, inp):
        self.yours = inp
    def give(self, amt):
        self.yours = self.yours + amt
        return self.yours
    def howmuch(self):
        return (A.yours, self.yours)

class B(A):
    yours = 0
    def give(self, amt):
        B.yours = B.yours + amt
        return A.howmuch(self)
    def take(self, amt):
        self.yours = self.yours - amt
        return self.yours
    def howmuch(self):
        return (B.yours, A.yours, self.yours)
```

Write the values of the following expressions. Write `None` when there is no value; write `Error` when an error results and explain briefly why it is an error. Assume that these expressions are evaluated one after another (all of the left column first, then right column).

```
  test = A(5)      test = B(5)
  test.take(2)    test.take(2)
  test.give(6)    test.give(6)
test.howmuch()  test.howmuch()
```

Ans:

- `None`: This makes `test` be equal to a new instance of `A`, with attribute `yours` equal to 5.
- `Error`: Instances of `A` don't have a `take` method.
- `11`: This adds 6 to the value of `test.yours` and returns its value.
- `(0, 11)`: At this point, we haven't changed the class attribute `A.yours`, so it still has value 0.
- `None`: This makes `test` be equal to a new instance of `B`; it inherits the init method from `A`, so at this point `test.yours` is equal to 5.
- `3`: Now, since `test` is an instance of `B`, the method `test.take` is defined; it changes `test.yours` to be 3 and returns its value.
- `(0, 3)`: Note that this refers to the `give` method defined in class `B`. So, first, the amount is used to increment `B.yours` to have value 6. Then, we return `A.yours` and `test.yours`, which are both unchanged.
- `(6, 0, 3)`: This just returns the current values of all three `yours` attributes, one in each class definition as well as `test.yours`.

3.7.6 Practice problem: The Best and the Brightest

Here are some class definitions, meant to represent a collection of students making up the student body of a prestigious university.

```
class Student:
    def __init__(self, name, iq, salary, height):
        self.name = name
        self.iq = iq
        self.salary = salary
        self.height = height

class StudentBody:
    def __init__(self):
        self.students = []
    def addStudent(self, student):
        self.students.append(student)
    def nameOfSmartest(self):
        # code will go here
    def funOfBest(self, fun, feature):
        # code will go here
```

The `StudentBody` class is missing the code for two methods:

- `nameOfSmartest`: returns the name attribute of the student with the highest IQ
- `funOfBest`: takes a procedure `fun` and a procedure `feature` as input, and returns the result of applying procedure `fun` to the student for whom the procedure `feature` yields the highest value

For the first two problems below, assume that these methods have been implemented.

Here is a student body:

```
jody = Student('Jody', 100, 100000, 80)
chris = Student('Chris', 150, 40000, 62)
dana = Student('Dana', 120, 2000, 70)
aardvarkU = StudentBody()
aardvarkU.addStudent(jody)
aardvarkU.addStudent(chris)
aardvarkU.addStudent(dana)
```

- What is the result of evaluating `aardvarkU.nameOfSmartest()`?

  Ans: 'Chris'

- Write a Python expression that will compute the name of the person who has the greatest value of IQ + height in aardvarkU (not just for the example student body above). You can define additional procedures if you need them.

  Ans: `aardvarkU.funOfBest(lambda x:  x.name, lambda x:  x.iq + x.height)`

  The second `lambda` expression specifies how to evaluate an element (that is, to sum its `iq` and `height` attributes) and the first `lambda` expression says what function to apply to the element with the highest score.

- Implement the `nameOfSmartest` method. For full credit, use `util.argmax` (defined below) or the `funOfBest` method.

  If `l` is a list of items and `f` is a procedure that maps an item into a numeric score, then `util.argmax(l, f)` returns the element of `l` that has the highest score.

  Ans:

```
def nameOfSmartest(self):
    return util.argmax(self.students, lambda x: x.iq).name

# or

def nameOfSmartest(self):
    return funOfBest(lambda x: x.name, lambda x: x.iq)
```

Note that in the first solution, the expression `util.argmax(self.students, lambda x: x.iq)` returns the object with the highest `iq` attribute; then we return the `name` attribute of that object.

- Implement the `funOfBest` method. For full credit, use `util.argmax`.

  Ans:

```
def funOfBest(self, fun, feature):
    return fun(util.argmax(self.students, feature))
```

3.7.7 Practice Problem: A Library with Class

Let's build a class to represent a library; let's call it `Library`. In this problem, we'll deal with some standard types of objects:

- A book is represented as a string – its title.
- A patron (person who uses the library) is represented as a string – his/her name.
- A date is represented by an integer – the number of days since the library opened.

The class should have an attribute called `dailyFine` that starts out as `0.25`. The class should have the following methods:

- `__init__`: takes a list of books and initializes the library.
- `checkOut`: is given a book, a patron and a date on which the book is being checked out and it records this. Each book can be kept for 7 days before it becomes overdue, i.e. if checked out on day x, it becomes due on day $x + 7$ and it will be considered overdue by one day on day $x + 8$. It returns `None`.
- `checkIn`: is given a book and a date on which the book is being returned and it updates the records. It returns a number representing the fine due if the book is overdue and 0.0 otherwise. The fine is the number of days overdue times the value of the `dailyFine` attribute.
- `overdueBooks`: is given a patron and a date and returns the list of books which that patron has checked out which are overdue at the given date.

Here is an example of the operation of the library:

```
>>> lib = Library(['a', 'b', 'c', 'd', 'e', 'f'])
>>> lib.checkOut('a', 'T', 1)
>>> lib.checkOut('c', 'T', 1)
>>> lib.checkOut('e', 'T', 10)
>>> lib.overdueBooks('T', 13)
['a', 'c']
>>> lib.checkIn('a', 13)
1.25
>>> lib.checkIn('c', 18)
2.50
>>> lib.checkIn('e', 18)
0.25
```

In the boxes below, define the `Library` class as described above. Above each answer box we repeat the specification for each of the attributes and methods given above. Make sure that you enter complete definitions in the boxes, including complete `class` and `def` statements.

Use a dictionary to store the contents of the library. Do not repeat code if at all possible. You can assume that all the operations are legal, for example, all books checked out are in the library and books checked in have been previously checked out.

Class definition:

Include both the start of the class definition and the method definition for `__init__` in this first answer box.

The class should have an attribute called `dailyFine` that starts out as `0.25`.

`__init__`: takes a list of books and initializes the library.

Ans:

```
class Library:
    dailyFine = 0.25
    def __init__(self, books):
        self.shelf = {}
        for book in books:
            self.shelf[book] = (None, None)  # (patron, dueDate)
```

The crucial part here is deciding what information to store. Here, for each book, we store a tuple of who has checked it out, and when it is due.

`checkOut`: is given a book, a patron and a date on which the book is being checked out and it records this. Each book can be kept for 7 days before it becomes overdue, i.e. if checked out on day x, it becomes due on day $x + 7$ and it will be considered overdue by one day on day $x + 8$. It returns `None`.

```
    def checkOut(self, book, patron, date):
        self.shelf[book] = (patron, date+7)
```

`checkIn`: is given a book and a date on which the book is being returned and it updates the records. It returns a number representing the fine due if the book is overdue and 0.0 otherwise. The fine is the number of days overdue times the value of the `dailyFine` attribute.

```
    def checkIn(self, book, date):
        patron, due = self.shelf[book]
        self.shelf[book] = (None, None)
        return max(0.0, (date - due))*self.dailyFine
```

Note the destructuring assignment in the first line. It's not crucial, but it's nice style, and keeps us from having to refer to components like `self.shelf[book][1]`, which are ugly, long, and hard to get right. Instead of using a `max`, you could use an `if` statement to handle the case of the book being overdue or not, but this is more compact and pretty clear.

`overdueBooks`: is given a patron and a date and returns the list of books which that patron has checked out which are overdue at the given date.

```
    def overdueBooks(self, patron, date):
        overdue = []
        for book in self.shelf:
            p, d = self.shelf[book]
            if p and d and p == patron and date > d:
                overdue.append(book)
        return overdue
```

It's not really necessary to check to be sure that `p` and `d` are not `None`, because `p == patron` will only be true for a real patron, and if the patron is not `None` then `d` won't be either. But this code makes it clear that we're only interested in books that have been checked out.

Define a new class called `LibraryGrace` that behaves just like the `Library` class except that it provides a grace period (some number of days after the actual due date) before fines start being accumulated. The number of days in the grace period is specified when an instance is created. See the example below.

```
>>> lib = LibraryGrace(2, ['a', 'b', 'c', 'd', 'e', 'f'])
>>> lib.checkOut('a', 'T', 1)
>>> lib.checkIn('a', 13)
0.75
```

Write the complete class definition for `LibraryGrace`. To get full credit you should not repeat any code that is already in the implementation of `Library`, in particular, you should not need to repeat the computation of the fine.

Class definition:

Ans:

```
class LibraryGrace(Library):
    def __init__(self, grace, books):
        self.grace = grace
        Library.__init__(self, books)
    def checkIn(self, book, date):
        return Library.checkIn(self, book, date - self.grace)
```

The crucial things here are: remembering to call the `__init__` method of the parent class and doing something to handle the grace period. In this example, when we check a book back in, we pretend we're actually checking it in on an earlier date. Alternatively, we could have changed the checkout method to pretend that the checkout was actually happening in the future.

Chapter 4

State Machines

State machines are a method of modeling systems whose output depends on the entire history of their inputs, and not just on the most recent input. Compared to purely functional systems, in which the output is purely determined by the input, state machines have a performance that is determined by its history. State machines can be used to model a wide variety of systems, including:

- user interfaces, with typed input, mouse clicks, etc.;
- conversations, in which, for example, the meaning of a word "it" depends on the history of things that have been said;
- the state of a spacecraft, including which valves are open and closed, the levels of fuel and oxygen, etc.; and
- the sequential patterns in DNA and what they mean.

State machine models can either be *continuous time* or *discrete time*. In continuous time models, we typically assume continuous spaces for the range of values of inputs and outputs, and use differential equations to describe the system's dynamics. This is an interesting and important approach, but it is hard to use it to describe the desired behavior of our robots, for example. The loop of reading sensors, computing, and generating an output is inherently discrete and too slow to be well-modeled as a continuous-time process. Also, our control policies are often highly non-linear and discontinuous. So, in this class, we will concentrate on discrete-time models, meaning models whose inputs and outputs are determined at specific increments of time, and which are synchronized to those specific time samples. Furthermore, in this chapter, we will make no assumptions about the form of the dependence of the output on the time-history of inputs; it can be an arbitrary function.

Generally speaking, we can think of the job of an embedded system as performing a *transduction* from a stream (infinite sequence) of input values to a stream of output values. In order to specify the behavior of a system whose output depends on the history of its inputs mathematically, you could think of trying to specify a mapping from $i_1, \ldots, i_t$ (sequences of previous inputs) to o_t (current output), but that could become very complicated to specify or execute as the history gets longer. In the state-machine approach, we try to find some set of *states* of the system, which capture the essential properties of the history of the inputs and are used to determine the current output of the system as well as its next state. It is an interesting and sometimes difficult modeling problem to find a good state-machine model with the right set of states; in this chapter we will explore how the ideas of PCAP can aid us in designing useful models.

One thing that is particularly interesting and important about state machine models is how many ways we can use them. In this class, we will use them in three fairly different ways:

1. **Synthetically**: State machines can specify a "program" for a robot or other system embedded in the world, with inputs being sensor readings and outputs being control commands.
2. **Analytically**: State machines can describe the behavior of the combination of a control system and the environment it is controlling; the input is generally a simple command to the entire system, and the output is some simple measure of the state of the system. The goal here is to analyze global properties of the coupled system, like whether it will converge to a steady state, or will oscillate, or will diverge.
3. **Predictively**: State machines can describe the way the environment works, for example, where I will end up if I drive down some road from some intersection. In this case, the inputs are control commands and the outputs are states of the external world. Such a model can be used to plan trajectories through the space of the external world to reach desirable states, by considering different courses of action and using the model to predict their results.

We will develop a single formalism, and an encoding of that formalism in Python classes, that will serve all three of these purposes.

Our strategy for building very complex state machines will be, abstractly, the same strategy we use to build any kind of complex machine. We will define a set of primitive machines (in this case, an infinite class of primitive machines) and then a collection of *combinators* that allow us to put primitive machines together to make more complex machines, which can themselves be abstracted and combined to make more complex machines.

4.1 Primitive State Machines

We can specify a transducer (a process that takes as input a sequence of values which serve as inputs to the state machine, and returns as ouput the set of outputs of the machine for each input) as a state machine (SM) by specifying:

- a set of *states*, S,
- a set of *inputs*, I, also called the *input vocabulary*,
- a set of *outputs*, O, also called the *output vocabulary*,

- a *next-state function*, $n(i_t, s_t) = s_{t+1}$, that maps the input at time t and the state at time t to the state at time $t + 1$,
- an *output function*, $o(i_t, s_t) = o_t$, that maps the input at time t and the state at time t to the output at time t; and
- an *initial state*, s_0, which is the state at time 0.

Here are a few state machines, to give you an idea of the kind of systems we are considering.

- A *tick-tock* machine that generates the sequence $1, 0, 1, 0, \ldots$ is a finite-state machine that ignores its input.
- The controller for a digital watch is a more complicated finite-state machine: it transduces a sequence of inputs (combination of button presses) into a sequence of outputs (combinations of segments illuminated in the display).
- The controller for a bank of elevators in a large office building: it transduces the current set of buttons being pressed and sensors in the elevators (for position, open doors, etc.) into commands to the elevators to move up or down, and open or close their doors.

The very simplest kind of state machine is a pure function: if the machine has no state, and the output function is purely a function of the input, for example, $o_t = i_t + 1$, then we have an immediate functional relationship between inputs and outputs on the same time step. Another simple class of SMs are *finite*-state machines, for which the set of possible states is finite. The elevator controller can be thought of as a finite-state machine, if elevators are modeled as being only at one floor or another (or possibly between floors); but if the controller models the exact position of the elevator (for the purpose of stopping smoothly at each floor, for example), then it will be most easily expressed using real numbers for the state (though any real instantiation of it can ultimately only have finite precision). A different, large class of SMs are describable as *linear, time-invariant* (LTI) systems. We will discuss these in detail in Chapter 5.

4.1.1 Examples

Let's look at several examples of state machines, with complete formal definitions.

4.1.1.1 Language Acceptor

Here is a finite-state machine whose output is *true* if the input string adheres to a simple pattern, and *false* otherwise. In this case, the pattern has to be $a, b, c, a, b, c, a, b, c, \ldots$. It uses the states 0, 1, and 2 to stand for the situations in which it is expecting an a, b, and c, respectively; and it uses state 3 for the situation in which it has seen an input that was not the one that was expected. Once the machine goes to state 3 (sometimes called a *rejecting state*), it never exits that state.

$$
\begin{aligned}
S &= \{0, 1, 2, 3\} \\
I &= \{a, b, c\} \\
O &= \{\mathit{true}, \mathit{false}\} \\
n(s, i) &= \begin{cases} 1 & \text{if } s = 0, i = a \\ 2 & \text{if } s = 1, i = b \\ 0 & \text{if } s = 2, i = c \\ 3 & \text{otherwise} \end{cases} \\
o(s, i) &= \begin{cases} \mathit{false} & \text{if } n(s, i) = 3 \\ \mathit{true} & \text{otherwise} \end{cases} \\
s_0 &= 0
\end{aligned}
$$

Figure 4.1 shows a state transition diagram for this state machine. Each circle represents a state. The arcs connecting the circles represent possible transitions the machine can make; the arcs are labeled with a pair i, o, which means that if the machine is in the state denoted by a circle with label s, and gets an input i, then the arc points to the next state, $n(s, i)$ and the output generated $o(s, i) = o$. Some arcs have several labels, indicating that there are many different inputs that will cause the same transition. Arcs can only be traversed in the direction of the arrow.

For a state transition diagram to be complete, there must be an arrow emerging from each state for each possible input i (if the next state is the same for some inputs, then we draw the graph more compactly by using a single arrow with multiple labels, as you will see below).

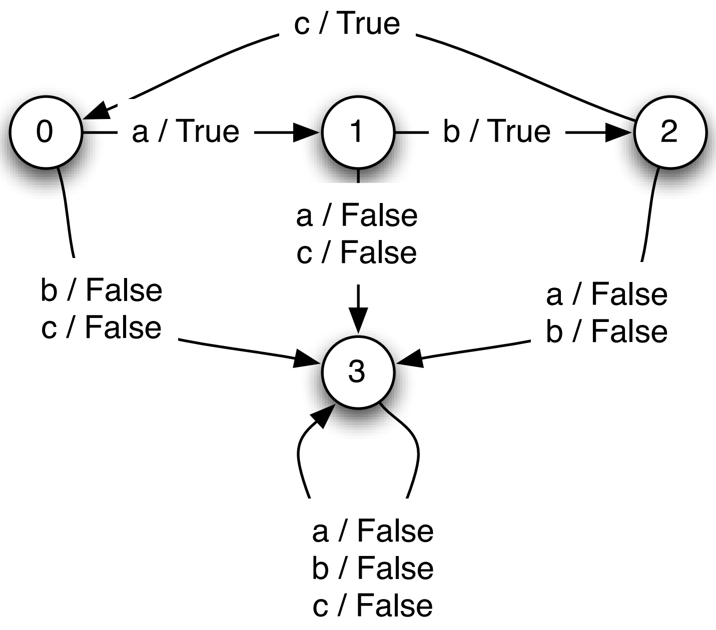


Figure 4.1: State transition diagram for language acceptor.

We will use tables like the following one to examine the evolution of a state machine:

time	0	1	2	...
input	i_0	i_1	i_2	...
state	s_0	s_1	s_2	...
output	o_0	o_1	o_2	...

For each column in the table, given the current input value and state we can use the output function o to determine the output in that column; and we use the n function applied to that input and state value to determine the state in the next column. Thus, just knowing the input sequence and s_0, and the next-state and output functions of the machine will allow you to fill in the rest of the table.

For example, here is the state of the machine at the initial time point:

time	0	1	2	...
input	i_0			...
state	s_0			...
output				...

Using our knowledge of the next state function n, we have:

time	0	1	2	...
input	i_0			...
state	s_0	s_1		...
output				...

and using our knowledge of the output function o, we have at the next input value

time	0	1	2	...
input	i_0	i_1		...
state	s_0	s_1		...
output	o_0			...

This completes one cycle of the statement machine, and we can now repeat the process.

Here is a table showing what the language acceptor machine does with input sequence ('a', 'b', 'c', 'a', 'c', 'a', 'b'):

time	0	1	2	3	4	5	6	7
input	'a'	'b'	'c'	'a'	'c'	'a'	'b'	
state	0	1	2	0	1	3	3	3
output	True	True	True	True	False	False	False	

The output sequence is (True, True, True, True, False, False, False).

Clearly we don't want to analyze a system by considering all input sequences, but this table helps us understand the state transitions of the system model.

To Learn More
Finite-state machine language acceptors can be built for a class of patterns called *regular languages*. There are many more complex patterns (such as the set of strings with equal numbers of 1's and 0's) that cannot be recognized by finite-state machines, but can be recognized by a specialized kind of infinite-state machine called a *stack machine*. To learn more about these fundamental ideas in *computability theory*, start with the Wikipedia article on **Computability**.

4.1.1.2 Up and Down Counter

This machine can count up and down; its state space is the countably infinite set of integers. It starts in state 0. Now, if it gets input u, it goes to state 1; if it gets u again, it goes to state 2. If it gets d, it goes back down to 1, and so on. For this machine, the output is always the same as the next state.

$$\begin{aligned}
S &= \textit{integers} \\
I &= \{u, d\} \\
O &= \textit{integers} \\
n(s,i) &= \begin{cases} s+1 & \text{if } i = u \\ s-1 & \text{if } i = d \end{cases} \\
o(s,i) &= n(s,i) \\
s_0 &= 0
\end{aligned}$$

Here is a table showing what the up and down counter does with input sequence u, u, u, d, d, u:

time	0	1	2	3	4	5	6
input	u	u	u	d	d	u	
state	0	1	2	3	2	1	2
output	1	2	3	2	1	2	

The output sequence is $1, 2, 3, 2, 1, 2$.

4.1.1.3 Delay

An even simpler machine just takes the input and passes it through to the output, but with one step of delay, so the kth element of the input sequence will be the $k+1$st element of the output sequence. Here is the formal machine definition:

$$\begin{aligned}
S &= \textit{anything} \\
I &= \textit{anything} \\
O &= \textit{anything} \\
n(s,i) &= i \\
o(s,i) &= s \\
s_0 &= 0
\end{aligned}$$

Given an input sequence $i_0, i_1, i_2, \ldots$, this machine will produce an output sequence $0, i_0, i_1, i_2, \ldots$. The initial 0 comes because it has to be able to produce an output before it has even seen an input, and that output is produced based on the initial state, which is 0. This very simple building block will come in handy for us later on.

Here is a table showing what the delay machine does with input sequence $3, 1, 2, 5, 9$:

time	0	1	2	3	4	5
input	3	1	2	5	9	
state	0	3	1	2	5	9
output	0	3	1	2	5	

The output sequence is $0, 3, 1, 2, 5$.

4.1.1.4 Accumulator

Here is a machine whose output is the sum of all the inputs it has ever seen.

$$\begin{aligned}
S &= \textit{numbers} \\
I &= \textit{numbers} \\
O &= \textit{numbers} \\
n(s,i) &= s + i \\
o(s,i) &= n(s,i) \\
s_0 &= 0
\end{aligned}$$

Here is a table showing what the accumulator does with input sequence $100, -3, 4, -123, 10$:

time	0	1	2	3	4	5
input	100	-3	4	-123	10	
state	0	100	97	101	-22	-12
output	100	97	101	-22	-12	

4.1.1.5 Average2

Here is a machine whose output is the average of the current input and the previous input. It stores its previous input as its state.

$$\begin{aligned} S &= numbers \\ I &= numbers \\ O &= numbers \\ n(s,i) &= i \\ o(s,i) &= (s+i)/2 \\ s_0 &= 0 \end{aligned}$$

Here is a table showing what the average2 machine does with input sequence $100, -3, 4, -123, 10$:

time	0	1	2	3	4	5
input	100	-3	4	-123	10	
state	0	100	-3	4	-123	10
output	50	48.5	0.5	-59.5	-56.5	

4.1.2 State Machines in Python

Now, it is time to make computational implementations of state machine models. In this section we will build up some basic Python infrastructure to make it straightforward to define primitive machines; in later sections we will see how to combine primitive machines into more complex structures.

We will use Python's object-oriented facilities to make this convenient. We have an abstract class, `SM`, which will be the superclass for all of the particular state machine classes we define. It does not make sense to make an instance of `SM`, because it does not actually specify the behavior of the machine; it just provides some utility methods. To specify a new type of state machine, you define a new class that has `SM` as a superclass.

In any subclass of `SM`, it is crucial to define an attribute `startState`, which specifies the initial state of the machine, and a method `getNextValues` which takes the state at time t and the input at time t as input, and returns the state at time $t+1$ and the output at time t. This is a choice that we have made as designers of our state machine model; we will rely on these two pieces of information in our underlying infrastructure for simulating state machines, as we will see shortly.

Here, for example, is the Python code for an accumulator state machine, which implements the definition given in Section 4.1.1.4[1].

[1]Throughout this code, we use `inp`, instead of `input`, which would be clearer. The reason is that the name `input` is used by Python as a function. Although it is legal to re-use it as the name of an argument

```
class Accumulator(SM):
    startState = 0
    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
```

It is important to note that `getNextValues` *does not change the state of the machine*, in other words, it does not mutate a state variable. Its job is to be a pure function: to answer the question of what the next state and output would be if the current state were `state` and the current input were `inp`. We will use the `getNextValues` methods of state machines later in the class to make plans by considering alternative courses of action, so they *must not have any side effects.* As we noted, this is our choice as designers of the state machine infrastructure. We could have chosen to implement things differently, however this choice will prove to be very useful. Thus, in all our state machines, the function `getNextValues` will capture the transition from input and state to output and state, without mutating any stored state values.

To run a state machine, we make an instance of the appropriate state-machine class, call its `start` method (a built in method we will see shortly) to set the state to the starting state, and then ask it to take steps; each step consists of generating an output value (which is printed) and updating the state to the next state, based on the input. The abstract superclass `SM` defines the `start` and `step` methods. These methods are in charge of actually initializing and then changing the state of the machine as it is being executed. They do this by calling the `getNextValues` method for the class to which this instance belongs. The `start` method takes no arguments (but, even so, we have to put parentheses after it, indicating that we want to call the method, not to do something with the method itself); the step method takes one argument, which is the input to the machine on the next step. So, here is a run of the accumulator, in which we feed it inputs 3, 4, and -2:

```
>>> a = Accumulator()
>>> a.start()
>>> a.step(3)
3
>>> a.step(4)
7
>>> a.step(-2)
5
```

The class `SM` specifies how state machines work in general; the class `Accumulator` specifies how accumulator machines work in general; and the instance `a` is a particular machine with a particular current state. We can make another instance of accumulator:

```
>>> b = Accumulator()
```

to a procedure, doing so is a source of bugs that are hard to find (if, for instance, you misspell the name `input` in the argument list, your references to `input` later in the procedure will be legal, but will return the built-in function.)

```
>>> b.start()
>>> b.step(10)
10
>>> b.state
10
>>> a.state
5
```

Now, we have two accumulators, `a`, and `b`, which remember, individually, in what states they exist. Figure 4.2 shows the class and instance structures that will be in place after creating these two accumulators.

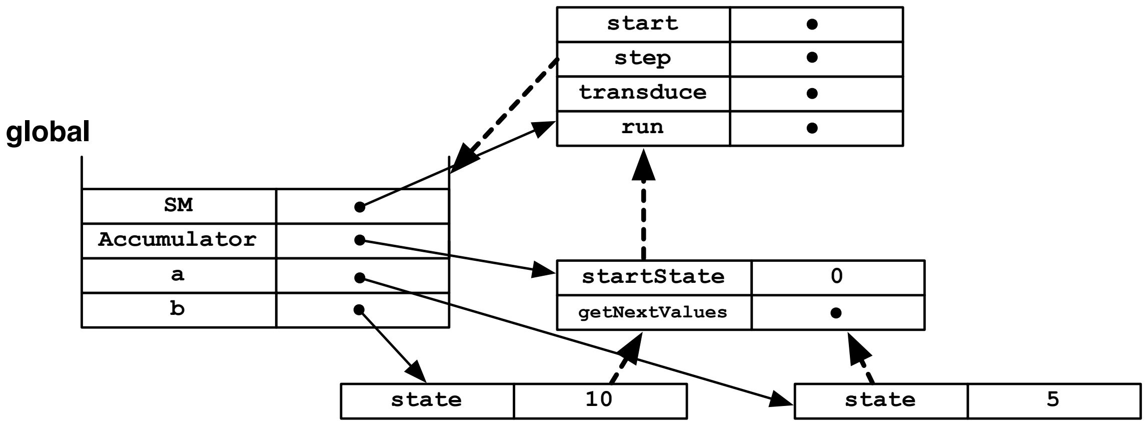


Figure 4.2: Classes and instances for an `Accumulator` `SM`. All of the dots represent procedure objects.

4.1.2.1 Defining a Type of SM

Let's go back to the definition of the `Accumulator` class, and study it piece by piece.

First, we define the class and say that it is a subclass of `SM`:

```
class Accumulator(SM):
```

Next, we define an attribute of the class, `startState`, which is the starting state of the machine. In this case, our accumulator starts up with the value 0.

```
    startState = 0
```

Note that `startState` is required by the underlying `SM` class, so we must either define it in our subclass definition or use a default value from the `SM` superclass.

The next method defines both the next-state function and the output function, by taking the current state and input as arguments and returning a tuple containing both the next state and the output.

For our accumulator, the next state is just the sum of the previous state and the input; and the output is the same thing:

```
    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
```

It is crucial that `getNextValues` *be a pure function; that is, that it not change the state of the object (by assigning to any attributes of* `self`*).* It must simply compute the necessary values and return them.

Sometimes, it is convenient to arrange it so that the class really defines a range of machines with slightly different behavior, which depends on some parameters that we set at the time we create an instance. So, for example, if we wanted to specify the initial value for an accumulator at the time the machine is created, we could add an `__init__` method[2] to our class, which takes an initial value as a parameter and uses it to set an attribute called `startState` of the instance.[3]

```
class Accumulator(SM):
    def __init__(self, initialValue):
        self.startState = initialValue
    def getNextValues(self, state, inp):
        return state + inp, state + inp
```

Now we can make an accumulator and run it like this:

```
>>> c = Accumulator(100)
>>> c.start()
>>> c.step(20)
120
>>> c.step(2)
122
```

4.1.2.2 The SM Class

The `SM` class contains generally useful methods that apply to all state machines. A state machine is an instance of any subclass of `SM`, that has defined the attribute `startState` and the method `getNextValues`, as we did for the `Accumulator` class. Here we examine these methods in more detail.

[2]Remember that the `__init__` method is a special feature of the Python object-oriented system, which is called whenever an instance of the associated class is created.

[3]Note that in the original version of `Accumulator`, `startState` was an attribute of the class, since it was the same for every instance of the class; now that we want it to be different for different instances, we need `startState` to be an attribute of the instance rather than the class, which is why we assign it in the `__init__` method, which modifies the already-created instance.

Running a machine
The first group of methods allows us to run a state machine. To run a machine is to provide it with a sequence of inputs and then sequentially go forward, computing the next state and generating the next output, as if we were filling in a state table.

To run a machine, we have to start by calling the `start` method. All it does is create an attribute of the instance, called `state`, and assign to it the value of the `startState` attribute. It is crucial that we have both of these attributes: if we were to just modify `startState`, then if we wanted to run this machine again, we would have permanently forgotten what the starting state should be. Note that `state` becomes a repository for the state of **this** instance; however we should not mutate it directly. This variable becomes the internal representation of state for each instance of this class.

```
class SM:
    def start(self):
        self.state = self.startState
```

Once it has started, we can ask it to take a step, using the `step` method, which, given an input, computes the output and updates the internal state of the machine, and returns the output value.

```
    def step(self, inp):
        (s, o) = self.getNextValues(self.state, inp)
        self.state = s
        return o
```

To run a machine on a whole sequence of input values, we can use the `transduce` method, which will return the sequence of output values that results from feeding the elements of the list `inputs` into the machine in order.

```
    def transduce(self, inputs):
        self.start()
        return [self.step(inp) for inp in inputs]
```

Here are the results of running `transduce` on our accumulator machine. We run it twice, first with a simple call that does not generate any debugging information, and simply returns the result. The second time, we ask it to be verbose, resulting in a print-out of what is happening on the intermediate steps. [4]

```
a = Accumulator()
>>> a.transduce([100, -3, 4, -123, 10])
[100, 97, 101, -22, -12]
>>> a.transduce([100, -3, 4, -123, 10], verbose = True)
```

[4]In fact, the second machine trace, and all the others in this section were generated with a call like:
`>>> a.transduce([100, -3, 4, -123, 10], verbose = True, compact = True)`

```
Start state: 0
In: 100 Out: 100 Next State: 100
In: -3 Out: 97 Next State: 97
In: 4 Out: 101 Next State: 101
In: -123 Out: -22 Next State: -22
In: 10 Out: -12 Next State: -12
[100, 97, 101, -22, -12]
```

Some machines do not take any inputs; in that case, we can simply call the `SM run` method, which is equivalent to doing `transduce` on an input sequence of `[None, None, None, ...]`.

```
    def run(self, n = 10):
        return self.transduce([None]*n)
```

Default Methods

In order to make the specifications for the simplest machine types as short as possible, we have also provided a set of default methods in the `SM` class. These default methods say that, unless they are overridden in a subclass, as they were when we defined `Accumulator`, we will assume that a machine starts in state `None` and that its output is the same as its next state.

```
    startState = None
    def getNextValues(self, state, inp):
        nextState = self.getNextState(state, inp)
        return (nextState, nextState)
```

Because these methods are provided in `SM`, we can define, for example, a state machine whose output is always `k` times its input, with this simple class definition, which defines a `getNextState` procedure that simply returns a value that is treated as both the next state and the output.

```
class Gain(SM):
    def __init__(self, k):
        self.k = k
    def getNextState(self, state, inp):
        return inp * self.k
```

We can use this class as follows:

```
>>> g = Gain(3)
>>> g.transduce([1.1, -2, 100, 5])
[3.3000000000000003, -6, 300, 15]
```

The parameter `k` is specified at the time the instance is created. Then, the output of the machine is always just `k` times the input.

We can also use this strategy to write the `Accumulator` class even more succinctly:

```
class Accumulator(SM):
    startState = 0
    def getNextState(self, state, inp):
        return state + inp
```

The output of the `getNextState` method will be treated both as the output and the next state of the machine, because the inherited `getNextValues` function uses it to compute both values.

4.1.2.3 More Examples

Here are Python versions of the rest of the machines we introduced in the first section.

Language Acceptor

Here is a Python class for a machine that "accepts" the language that is any prefix of the infinite sequence `['a', 'b', 'c', 'a', 'b', 'c', ....]`.

```
class ABC(SM):
    startState = 0
    def getNextValues(self, state, inp):
        if state == 0 and inp == 'a':
            return (1, True)
        elif state == 1 and inp == 'b':
            return (2, True)
        elif state == 2 and inp == 'c':
            return (0, True)
        else:
            return (3, False)
```

It behaves as we would expect. As soon as it sees a character that deviates from the desired sequence, the output is `False`, and it will remain `False` for ever after.

```
>>> abc = ABC()
>>> abc.transduce(['a','a','a'], verbose = True)
Start state: 0
In: a Out: True Next State: 1
In: a Out: False Next State: 3
In: a Out: False Next State: 3
[True, False, False]
```

```
>>> abc.transduce(['a', 'b', 'c', 'a', 'c', 'a', 'b'], verbose = True)
Start state: 0
In: a Out: True Next State: 1
In: b Out: True Next State: 2
In: c Out: True Next State: 0
In: a Out: True Next State: 1
In: c Out: False Next State: 3
In: a Out: False Next State: 3
In: b Out: False Next State: 3
[True, True, True, True, False, False, False]
```

Count Up and Down

This is a direct translation of the machine defined in Section 4.1.1.2.

```
class UpDown(SM):
    startState = 0
    def getNextState(self, state, inp):
        if inp == 'u':
            return state + 1
        else:
            return state - 1
```

We take advantage of the default `getNextValues` method to make the output the same as the next state.

```
>>> ud = UpDown()
>>> ud.transduce(['u','u','u','d','d','u'], verbose = True)
Start state: 0
In: u Out: 1 Next State: 1
In: u Out: 2 Next State: 2
In: u Out: 3 Next State: 3
In: d Out: 2 Next State: 2
In: d Out: 1 Next State: 1
In: u Out: 2 Next State: 2
[1, 2, 3, 2, 1, 2]
```

Delay

In order to make a machine that delays its input stream by one time step, we have to specify what the first output should be. We do this by passing the parameter, `v0`, into the `__init__` method of the `Delay` class. The state of a `Delay` machine is just the input from the previous step, and the output is the state (which is, therefore, the input from the previous time step).

```
class Delay(SM):
    def __init__(self, v0):
        self.startState = v0
    def getNextValues(self, state, inp):
        return (inp, state)

>>> d = Delay(7)
>>> d.transduce([3, 1, 2, 5, 9], verbose = True)
Start state: 7
In: 3 Out: 7 Next State: 3
In: 1 Out: 3 Next State: 1
In: 2 Out: 1 Next State: 2
In: 5 Out: 2 Next State: 5
In: 9 Out: 5 Next State: 9
[7, 3, 1, 2, 5]

>>> d100 = Delay(100)
>>> d100.transduce([3, 1, 2, 5, 9], verbose = True)
Start state: 100
In: 3 Out: 100 Next State: 3
In: 1 Out: 3 Next State: 1
In: 2 Out: 1 Next State: 2
In: 5 Out: 2 Next State: 5
In: 9 Out: 5 Next State: 9
[100, 3, 1, 2, 5]
```

We will use this machine so frequently that we put its definition in the `sm` module (file), along with the class.

R

We can use `R` as another name for the `Delay` class of state machines. It will be an important primitive in a compositional system of *linear time-invariant systems*, which we explore in the next chapter.

Average2

Here is a state machine whose output at time t is the average of the input values from times $t - 1$ and t.

```
class Average2(SM):
    startState = 0
    def getNextValues(self, state, inp):
        return (inp, (inp + state) / 2.0)
```

It needs to remember the previous input, so the next state is equal to the input. The output is the average of the current input and the state (because the state is the previous input).

```
>>> a2 = Average2()
>>> a2.transduce([10, 5, 2, 10], verbose = True, compact = True)
Start state: 0
In: 10 Out: 5.0 Next State: 10
In: 5 Out: 7.5 Next State: 5
In: 2 Out: 3.5 Next State: 2
In: 10 Out: 6.0 Next State: 10
[5.0, 7.5, 3.5, 6.0]
```

Sum of Last Three Inputs

Here is an example of a state machine where the state is actually a list of values. Generally speaking, the state can be anything (a dictionary, a list, etc.); but it is important to be sure that the `getNextValues` method does not make direct changes to components of the state, instead returning a *new copy* of the state with appropriate changes. We may make several calls to the `getNextValues` function on one step (or, later in our work, call the `getNextValues` function with several different inputs to see what would happen under different choices); these function calls are made to find out a value of the next state, but if they actually change the state, then the same call with the same arguments may return a different value the next time.

This machine generates as output at time t the sum of i_{t-2}, i_{t-1} and i_t; that is, of the last three inputs. In order to do this, it has to remember the values of two previous inputs; so the state is a pair of numbers. We have defined it so that the initial state is `(0, 0)`. The `getNextValues` method gets rid of the oldest value that it has been remembering, and remembers the current input as part of the state; the output is the sum of the current input with the two old inputs that are stored in the state. Note that the first line of the `getNextValues` procedure is a structured assignment (see Section 3.3).

```
class SumLast3 (SM):
    startState = (0, 0)
    def getNextValues(self, state, inp):
        (previousPreviousInput, previousInput) = state
        return ((previousInput, inp),
                previousPreviousInput + previousInput + inp)

>>> sl3 = SumLast3()
>>> sl3.transduce([2, 1, 3, 4, 10, 1, 2, 1, 5], verbose = True)
Start state: (0, 0)
In: 2 Out: 2 Next State: (0, 2)
In: 1 Out: 3 Next State: (2, 1)
In: 3 Out: 6 Next State: (1, 3)
In: 4 Out: 8 Next State: (3, 4)
In: 10 Out: 17 Next State: (4, 10)
In: 1 Out: 15 Next State: (10, 1)
In: 2 Out: 13 Next State: (1, 2)
In: 1 Out: 4 Next State: (2, 1)
```

```
In: 5 Out: 8 Next State: (1, 5)
[2, 3, 6, 8, 17, 15, 13, 4, 8]
```

Selector

A simple functional machine that is very useful is the `Select` machine. You can make many different versions of this, but the simplest one takes an input that is a stream of lists or tuples of several values (or structures of values) and generates the stream made up only of the `k`th elements of the input values. Which particular component this machine is going to select is determined by the value `k`, which is passed in at the time the machine instance is initialized.

```
class Select (SM):
    def __init__(self, k):
        self.k = k
    def getNextState(self, state, inp):
        return inp[self.k]
```

4.1.3 Simple Parking Gate Controller

As one more demonstration, here is a simple example of a finite-state controller for a gate leading out of a parking lot.

The gate has three sensors:

- `gatePosition` has one of three values `'top'`, `'middle'`, `'bottom'`, signifying the position of the arm of the parking gate.
- `carAtGate` is `True` if a car is waiting to come through the gate and `False` otherwise.
- `carJustExited` is `True` if a car has just passed through the gate; it is true for only one step before resetting to `False`.

> Pause to try
> How many possible inputs are there?
>
> A: 12.

The gate has three possible outputs (think of them as controls to the motor for the gate arm): `'raise'`, `'lower'`, and `'nop'`. (Nop means“no operation.”)

Roughly, here is what the gate needs to do:

- If a car wants to come through, the gate needs to raise the arm until it is at the top position.
- Once the gate is at the top position, it has to stay there until the car has driven through the gate.
- After the car has driven through, the gate needs to lower the arm until it reaches the bottom position.

So, we have designed a simple finite-state controller with a state transition diagram as shown in Figure 4.3. The machine has four possible states: `'waiting'` (for a car to arrive at the gate), `'raising'` (the arm), `'raised'` (the arm is at the top position and we're waiting for the car to drive through the gate), and `'lowering'` (the arm). To keep the figure from being too cluttered, we do not label each arc with every possible input that would cause that transition to be made: instead, we give a condition (think of it as a Boolean expression) that, if true, would cause the transition to be followed. The conditions on the arcs leading out of each state cover all the possible inputs, so the machine remains completely well specified.

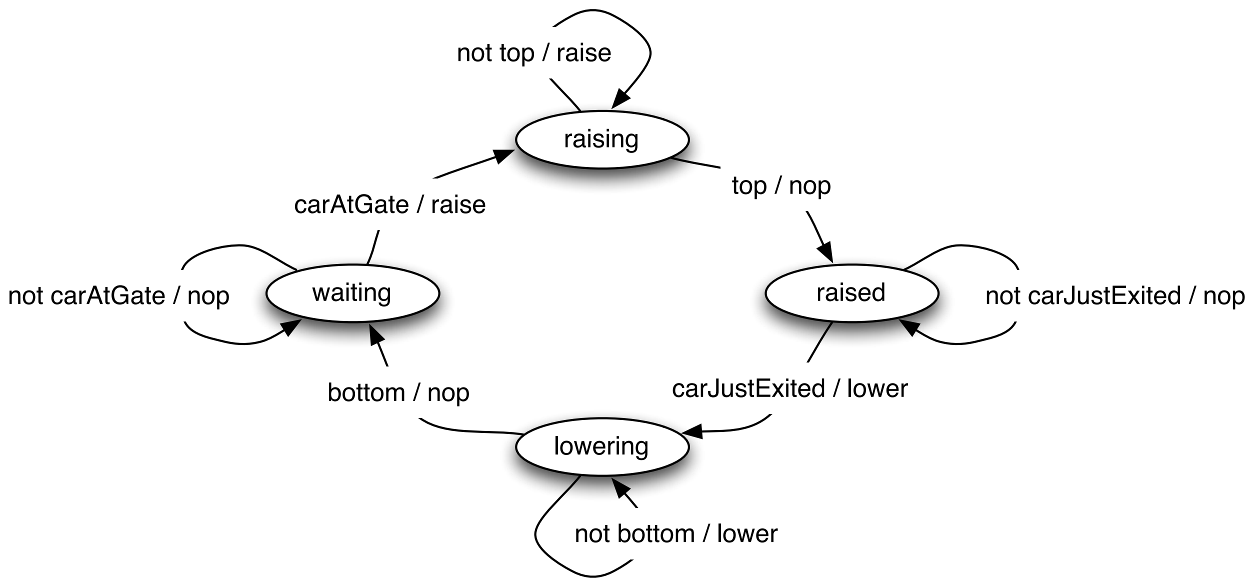


Figure 4.3: State transition diagram for parking gate controller.

Here is a simple state machine that implements the controller for the parking gate. For compactness, the `getNextValues` method starts by determining the next state of the gate. Then, depending on the next state, the `generateOutput` method selects the appropriate output.

```
class SimpleParkingGate (SM):
    startState = 'waiting'

    def generateOutput(self, state):
        if state == 'raising':
            return 'raise'
        elif state == 'lowering':
            return 'lower'
        else:
            return 'nop'

    def getNextValues(self, state, inp):
        (gatePosition, carAtGate, carJustExited) = inp
        if state == 'waiting' and carAtGate:
            nextState = 'raising'
        elif state == 'raising' and gatePosition == 'top':
            nextState = 'raised'
        elif state == 'raised' and carJustExited:
            nextState = 'lowering'
        elif state == 'lowering' and gatePosition == 'bottom':
            nextState = 'waiting'
        else:
            nextState = state
        return (nextState, self.generateOutput(nextState))
```

In the situations where the state does not change (that is, when the arcs lead back to the same state in the diagram), we do not explicitly specify the next state in the code: instead, we cover it in the `else` clause, saying that unless otherwise specified, the state stays the same. So, for example, if the state is `raising` but the `gatePosition` is not yet `top`, then the state simply stays `raising` until the top is reached.

```
>>> spg = SimpleParkingGate()
>>> spg.transduce(testInput, verbose = True)
Start state: waiting
In: ('bottom', False, False) Out: nop Next State: waiting
In: ('bottom', True, False) Out: raise Next State: raising
In: ('bottom', True, False) Out: raise Next State: raising
In: ('middle', True, False) Out: raise Next State: raising
In: ('middle', True, False) Out: raise Next State: raising
In: ('middle', True, False) Out: raise Next State: raising
In: ('top', True, False) Out: nop Next State: raised
In: ('top', True, False) Out: nop Next State: raised
In: ('top', True, False) Out: nop Next State: raised
In: ('top', True, True) Out: lower Next State: lowering
In: ('top', True, True) Out: lower Next State: lowering
In: ('top', True, False) Out: lower Next State: lowering
In: ('middle', True, False) Out: lower Next State: lowering
In: ('middle', True, False) Out: lower Next State: lowering
In: ('middle', True, False) Out: lower Next State: lowering
In: ('bottom', True, False) Out: nop Next State: waiting
In: ('bottom', True, False) Out: raise Next State: raising
['nop', 'raise', 'raise', 'raise', 'raise', 'raise', 'nop', 'nop',
 'nop', 'lower', 'lower', 'lower', 'lower', 'lower', 'lower', 'nop', 'raise']
```

Exercise 4.1
What would the code for this machine look like if it were written without using the `generateOutput` method?

4.2 Basic Combination and Abstraction of State Machines

In the previous section, we studied the definition of a primitive state machine, and saw a number of examples. State machines are useful for a wide variety of problems, but specifying complex machines by explicitly writing out their state transition functions can be quite tedious. Ultimately, we will want to build large state-machine descriptions compositionally, by specifying primitive machines and then combining them into more complex systems. We will start here by looking at ways of combining state machines.

We can apply our PCAP (primitive, combination, abstraction, pattern) methodology, to build more complex SMs out of simpler ones. In the rest of this section we consider "dataflow" compositions, where inputs and outputs of primitive machines are connected together; after that, we consider "conditional" compositions that make use of different sub-machines depending on the input to the machine, and finally "sequential" compositions that run one machine after another.

4.2.1 Cascade Composition

In cascade composition, we take two machines and use the output of the first one as the input to the second, as shown in Figure 4.4. The result is a new composite machine, whose input vocabulary is the input vocabulary of the first machine and whose output vocabulary is the output vocabulary of the second machine. It is, of course, crucial that the output vocabulary of the first machine be the same as the input vocabulary of the second machine.

Recalling the `Delay` machine from the previous section, let's see what happens if we make the cascade composition of two delay machines. Let m_1 be a delay machine with initial value 99 and m_2 be a delay machine with initial value 22. Then $Cascade(m_1, m_2)$ is a new state machine, constructed by making the output of m_1 be the input of m_2. Now, imagine we feed a sequence of values, $3, 8, 2, 4, 6, 5$, into the composite machine, m. What will come out? Let's try to understand this by making a table of the states and values at different times:

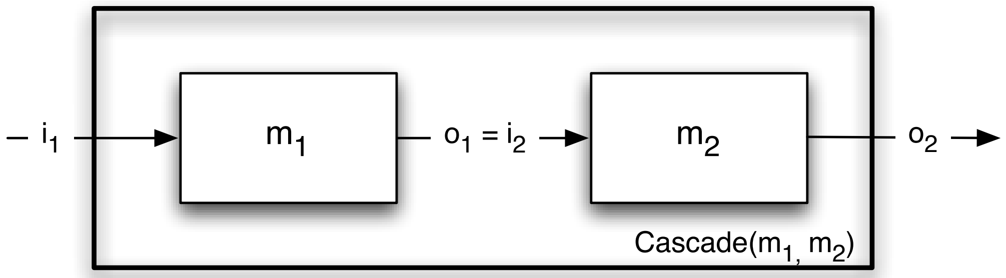


Figure 4.4: Cascade composition of state machines.

time	0	1	2	3	4	5	6
m_1 input	3	8	2	4	6	5	
m_1 state	99	3	8	2	4	6	5
m_1 output	99	3	8	2	4	6	
m_2 input	99	3	8	2	4	6	
m_2 state	22	99	3	8	2	4	6
m_2 output	22	99	3	8	2	4	

The output sequence is $22, 99, 3, 8, 2, 4$, which is the input sequence, delayed by two time steps.

Another way to think about cascade composition is as follows. Let the input to m_1 at time t be called $i_1[t]$ and the output of m_1 at time t be called $o_1[t]$. Then, we can describe the workings of the delay machine in terms of an equation:

$$\begin{aligned} o_1[t] &= i_1[t-1] \text{ for all values of } t > 0; \\ o_1[0] &= init_1 \end{aligned}$$

that is, that the output value at every time t is equal to the input value at the previous time step. You can see that in the table above. The same relation holds for the input and output of m_2:

$$\begin{aligned} o_2[t] &= i_2[t-1] \text{ for all values of } t > 0. \\ o_2[0] &= init_2 \end{aligned}$$

Now, since we have connected the output of m_1 to the input of m_2, we also have that $i_2[t] = o_1[t]$ for all values of t. This lets us make the following derivation:

$$\begin{aligned} o_2[t] &= i_2[t-1] \\ &= o_1[t-1] \\ &= i_1[t-2] \end{aligned}$$

This makes it clear that we have built a "delay by two" machine, by cascading two single delay machines.

As with all of our systems of combination, we will be able to form the cascade composition not only of two primitive machines, but of any two machines that we can make, through any set of compositions of primitive machines.

Here is the Python code for an `Increment` machine. It is a pure function whose output at time *t* is just the input at time *t* plus the constant `incr`. The `safeAdd` function is the same as addition, if the inputs are numbers. We will see, later, why it is important.

```
class Increment(SM):
    def __init__(self, incr):
        self.incr = incr
    def getNextState(self, state, inp):
        return safeAdd(inp, self.incr)
```

Exercise 4.2
Derive what happens when you cascade two delay-by-two machines?

Exercise 4.3
What is the difference between these two machines?

```
>>> foo1 = sm.Cascade(sm.Delay(100), Increment(1))
>>> foo2 = sm.Cascade(Increment(1), sm.Delay(100))
```

Demonstrate by drawing a table of their inputs, states, and outputs, over time.

4.2.2 Parallel Composition

In parallel composition, we take two machines and run them "side by side". They both take the same input, and the output of the composite machine is the pair of outputs of the individual machines. The result is a new composite machine, whose input vocabulary is the same as the input vocabulary of the component machines (which is the same for both machines) and whose output vocabulary is pairs of elements, the first from the output vocabulary of the first machine and the second from the output vocabulary of the second machine. Figure 4.5 shows two types of parallel composition; in this section we are talking about the first type.

In Python, we can define a new class of state machines, called `Parallel`, which is a subclass of `SM`. To make an instance of `Parallel`, we pass two `SM`s of any type into the initializer. The state of the parallel machine is a pair consisting of the states of the constituent machines. So, the starting state is the pair of the starting states of the constituents.

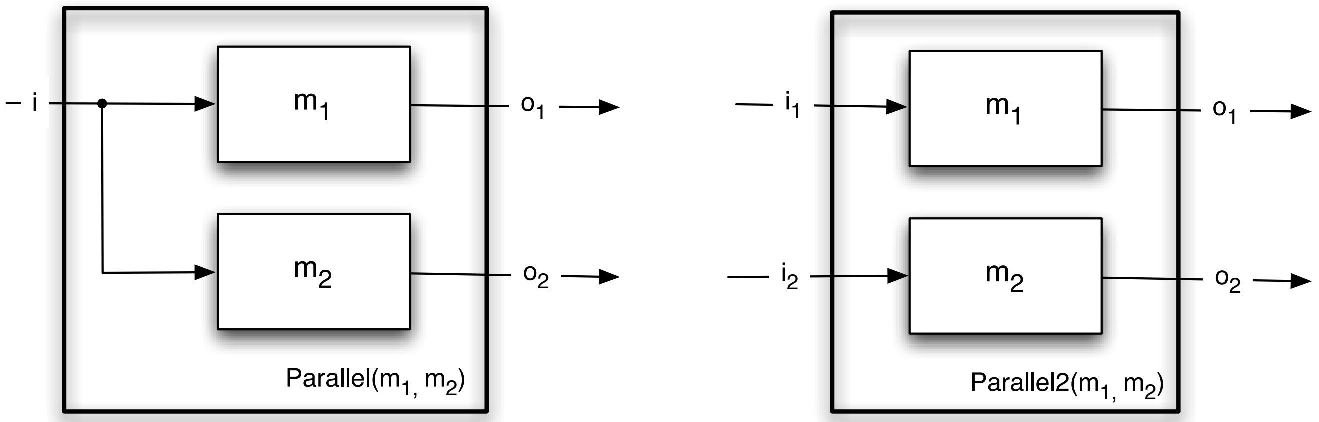


Figure 4.5: Parallel and Parallel2 compositions of state machines.

```
class Parallel (SM):
    def __init__(self, sm1, sm2):
        self.m1 = sm1
        self.m2 = sm2
        self.startState = (sm1.startState, sm2.startState)
```

To get a new state of the composite machine, we just have to get new states for each of the constituents, and return the pair of them; similarly for the outputs.

```
    def getNextValues(self, state, inp):
        (s1, s2) = state
        (newS1, o1) = self.m1.getNextValues(s1, inp)
        (newS2, o2) = self.m2.getNextValues(s2, inp)
        return ((newS1, newS2), (o1, o2))
```

4.2.2.1 Parallel2

Sometimes we will want a variant on parallel combination, in which rather than having the input be a single item which is fed to both machines, the input is a pair of items, the first of which is fed to the first machine and the second to the second machine. This composition is shown in the second part of Figure 4.5.

Here is a Python class that implements this two-input parallel composition. It can inherit the `__init__` method from `Parallel`, so use `Parallel` as the superclass, and we only have to define two methods.

```
class Parallel2 (Parallel):
    def getNextValues(self, state, inp):
        (s1, s2) = state
        (i1, i2) = splitValue(inp)
        (newS1, o1) = self.m1.getNextValues(s1, i1)
        (newS2, o2) = self.m2.getNextValues(s2, i2)
        return ((newS1, newS2), (o1, o2))
```

Later, when dealing with feedback systems (Section 4.2.3), we will need to be able to deal with `'undefined'` as an input. If the `Parallel2` machine gets an input of `'undefined'`, then we want to pass `'undefined'` into the constituent machines. We make our code more beautiful by defining the helper function below, which is guaranteed to return a pair, if its argument is either a pair or `'undefined'`.[5]

```
def splitValue(v):
    if v == 'undefined':
        return ('undefined', 'undefined')
    else:
        return v
```

4.2.2.2 ParallelAdd

The `ParallelAdd` state machine combination is just like `Parallel`, except that it has a single output whose value is the sum of the outputs of the constituent machines. It is straightforward to define:

```
class ParallelAdd (Parallel):
    def getNextValues(self, state, inp):
        (s1, s2) = state
        (newS1, o1) = self.m1.getNextValues(s1, inp)
        (newS2, o2) = self.m2.getNextValues(s2, inp)
        return ((newS1, newS2), o1 + o2)
```

4.2.3 Feedback Composition

Another important means of combination that we will use frequently is the feedback combinator, in which the output of a machine is fed back to be the input of the same machine at the next step, as shown in Figure 4.6. The first value that is fed back is the output associated with the initial state of the machine on which we are operating. It is crucial that the input and output vocabularies of the machine are the same (because the output at step t will be the input at step $t + 1$). Because we have fed the output back to the input, this machine does not consume any inputs; but we will treat the feedback value as an output of this machine.

Here is an example of using feedback to make a machine that counts. We can start with a simple machine, an incrementer, that takes a number as input and returns that same

[5]We are trying to make the code examples we show here as simple and clear as possible; if we were writing code for actual deployment, we would check and generate error messages for all sorts of potential problems (in this case, for instance, if `v` is neither `None` nor a two-element list or tuple.)

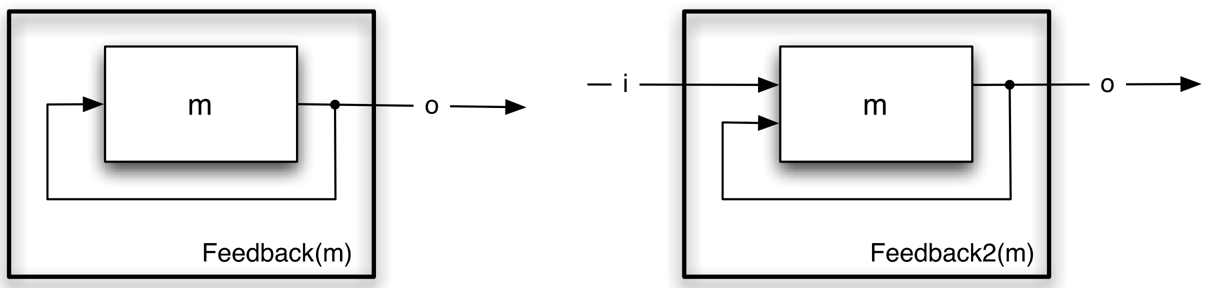


Figure 4.6: Two forms of feedback composition.

number plus 1 as the output. By itself, it has no memory. Here is its formal description:

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \\ O &= \textit{numbers} \\ n(s,i) &= i+1 \\ o(s,i) &= n(s,i) \\ s_0 &= 0 \end{aligned}$$

What would happen if we performed the feedback operation on this machine? We can try to understand this in terms of the input/output equations. From the definition of the increment machine, we have

$$o[t] = i[t] + 1 \ .$$

And if we connect the input to the output, then we will have

$$i[t] = o[t] \ .$$

And so, we have a problem; these equations cannot be satisfied.

A crucial requirement for applying feedback to a machine is: *that machine must not have a direct dependence of its output on its input.*

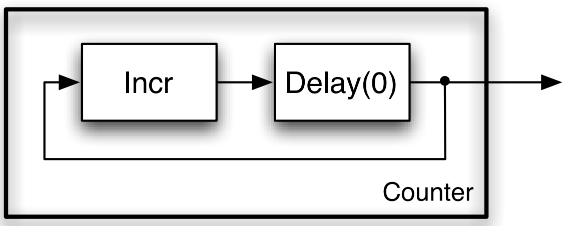


Figure 4.7: Counter made with feedback and serial combination of an incrementer and a delay.

We have already explored a `Delay` machine, which delays its output by one step. We can delay the result of our incrementer, by cascading it with a `Delay` machine, as shown in

Figure 4.7. Now, we have the following equations describing the system:

$$\begin{aligned} o_i[t] &= i_i[t] + 1 \\ o_d[t] &= i_d[t-1] \\ i_i[t] &= o_d[t] \\ i_d[t] &= o_i[t] \end{aligned}$$

The first two equations describe the operations of the increment and delay boxes; the second two describe the wiring between the modules. Now we can see that, in general,

$$\begin{aligned} o_i[t] &= i_i[t] + 1 \\ o_i[t] &= o_d[t] + 1 \\ o_i[t] &= i_d[t-1] + 1 \\ o_i[t] &= o_i[t-1] + 1 \end{aligned}$$

that is, that the output of the incrementer is going to be one greater on each time step.

Exercise 4.4
How could you use feedback and a *negation* primitive machine (which is a pure function that takes a Boolean as input and returns the negation of that Boolean) to make a machine whose output alternates between *true* and *false*.

4.2.3.1 Python Implementation

The following is a Python implementation of the feedback combinator, as a new subclass of `SM` that takes, at initialization time, a state machine.

```
class Feedback (SM):
    def __init__(self, sm):
        self.m = sm
        self.startState = self.m.startState
```

The starting state of the feedback machine is just the state of the constituent machine.

Generating an output for the feedback machine is interesting: by our hypothesis that the output of the constituent machine cannot depend directly on the current input, it means that, for the purposes of generating the output, we can actually feed an explicitly undefined value into the machine as input. Why would we do this? The answer is that we do not know what the input value should be (in fact, it is defined to be the output that we are trying to compute).

We must, at this point, add an extra condition on our `getNextValues` methods. They have to be prepared to accept `'undefined'` as an input. If they get an undefined input, they should return `'undefined'` as an output. For convenience, in our files, we have defined the procedures `safeAdd` and `safeMul` to do addition and multiplication, but passing through `'undefined'` if it occurs in either argument.

So: if we pass `'undefined'` into the constituent machine's `getNextValues` method, we must not get `'undefined'` back as output; if we do, it means that there is an immediate dependence of the output on the input. Now we know the output `o` of the machine.

To get the next state of the machine, we get the next state of the constituent machine, by taking the feedback value, `o`, that we just computed and using it as input for `getNextValues`. This will generate the next state of the feedback machine. (Note that throughout this process `inp` is ignored—a feedback machine has no input.)

```
    def getNextValues(self, state, inp):
        (ignore, o) = self.m.getNextValues(state, 'undefined')
        (newS, ignore) = self.m.getNextValues(state, o)
        return (newS, o)
```

Now, we can construct the counter we designed. The `Increment` machine, as we saw in its definition, uses a `safeAdd` procedure, which has the following property: if either argument is `'undefined'`, then the answer is `'undefined'`; otherwise, it is the sum of the inputs.

```
def makeCounter(init, step):
    return sm.Feedback(sm.Cascade(Increment(step), sm.Delay(init)))

>>> c = makeCounter(3, 2)
>>> c.run(verbose = True)
Start state: (None, 3)
Step: 0
 Feedback_96
     Cascade_97
         Increment_98 In: 3 Out: 5 Next State: 5
         Delay_99 In: 5 Out: 3 Next State: 5
Step: 1
 Feedback_96
     Cascade_97
         Increment_98 In: 5 Out: 7 Next State: 7
         Delay_99 In: 7 Out: 5 Next State: 7
Step: 2
 Feedback_96
     Cascade_97
         Increment_98 In: 7 Out: 9 Next State: 9
         Delay_99 In: 9 Out: 7 Next State: 9
Step: 3
 Feedback_96
     Cascade_97
         Increment_98 In: 9 Out: 11 Next State: 11
         Delay_99 In: 11 Out: 9 Next State: 11
Step: 4
 Feedback_96
     Cascade_97
         Increment_98 In: 11 Out: 13 Next State: 13
         Delay_99 In: 13 Out: 11 Next State: 13
...

[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

(The numbers, like `96` in `Feedback_96` are not important; they are just tags generated internally to indicate different instances of a class.)

Exercise 4.5
Draw state tables illustrating whether the following machines are different, and if so, how:

```
m1 = sm.Feedback(sm.Cascade(sm.Delay(1),Increment(1)))
m2 = sm.Feedback(sm.Cascade(Increment(1), sm.Delay(1)))
```

4.2.3.2 Fibonacci

Now, we can get very fancy. We can generate the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21, etc), in which the first two outputs are 1, and each subsequent output is the sum of the two previous outputs, using a combination of very simple machines. Basically, we have to arrange for the output of the machine to be fed back into a parallel combination of elements, one of which delays the value by one step, and one of which delays by two steps. Then, those values are added, to compute the next output. Figure 4.8 shows a diagram of one way to construct this system.

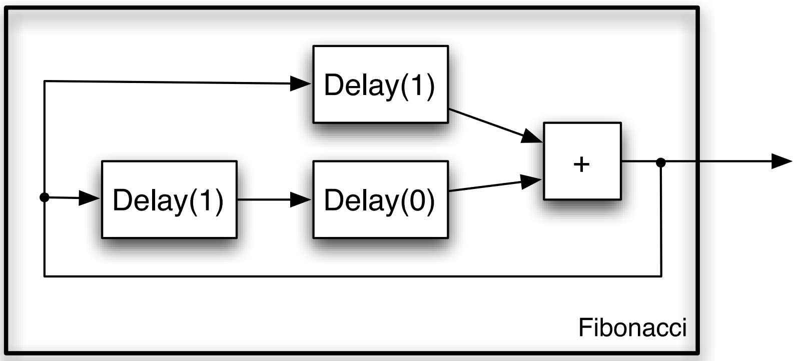


Figure 4.8: Machine to generate the Fibonacci sequence.

The corresponding Python code is shown below. First, we have to define a new component machine. An `Adder` takes pairs of numbers (appearing simultaneously) as input, and immediately generates their sum as output.

```
class Adder(SM):
    def getNextState(self, state, inp):
        (i1, i2) = splitValue(inp)
        return safeAdd(i1, i2)
```

Now, we can define our `fib` machine. It is a great example of building a complex machine out of very nearly trivial components. In fact, we will see in the next module that there is an interesting and important class of machines that can be constructed with cascade and parallel compositions of delay, adder, and gain machines. It is crucial for the delay machines to have the right values (as shown in the figure) in order for the sequence to start off correctly.

```
>>> fib = sm.Feedback(sm.Cascade(sm.Parallel(sm.Delay(1),
                                             sm.Cascade(sm.Delay(1), sm.Delay(0))),
                                 Adder()))

>>> fib.run(verbose = True)
Start state: ((1, (1, 0)), None)
Step: 0
 Feedback_100
     Cascade_101
         Parallel_102
             Delay_103 In: 1 Out: 1 Next State: 1
             Cascade_104
                 Delay_105 In: 1 Out: 1 Next State: 1
                 Delay_106 In: 1 Out: 0 Next State: 1
         Adder_107 In: (1, 0) Out: 1 Next State: 1
Step: 1
 Feedback_100
     Cascade_101
         Parallel_102
             Delay_103 In: 2 Out: 1 Next State: 2
             Cascade_104
                 Delay_105 In: 2 Out: 1 Next State: 2
                 Delay_106 In: 1 Out: 1 Next State: 1
         Adder_107 In: (1, 1) Out: 2 Next State: 2
Step: 2
 Feedback_100
     Cascade_101
         Parallel_102
             Delay_103 In: 3 Out: 2 Next State: 3
             Cascade_104
```

```
                 Delay_105 In: 3 Out: 2 Next State: 3
                 Delay_106 In: 2 Out: 1 Next State: 2
         Adder_107 In: (2, 1) Out: 3 Next State: 3
Step: 3
 Feedback_100
     Cascade_101
         Parallel_102
             Delay_103 In: 5 Out: 3 Next State: 5
             Cascade_104
                 Delay_105 In: 5 Out: 3 Next State: 5
                 Delay_106 In: 3 Out: 2 Next State: 3
         Adder_107 In: (3, 2) Out: 5 Next State: 5
...

[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Exercise 4.6
What would we have to do to this machine to get the sequence `[1, 1, 2, 3, 5, ...]`?

Exercise 4.7
Define fib as a composition involving only two delay components and an adder. You might want to use an instance of the `Wire` class.

A `Wire` is the completely passive machine, whose output is always instantaneously equal to its input. It is not very interesting by itself, but sometimes handy when building things.

```
class Wire(SM):
    def getNextState(self, state, inp):
        return inp
```

Exercise 4.8
Use feedback and a multiplier (analogous to `Adder`) to make a machine whose output doubles on every step.

Exercise 4.9
Use feedback and a multiplier (analogous to `Adder`) to make a machine whose output squares on every step.

4.2.3.3 Feedback2

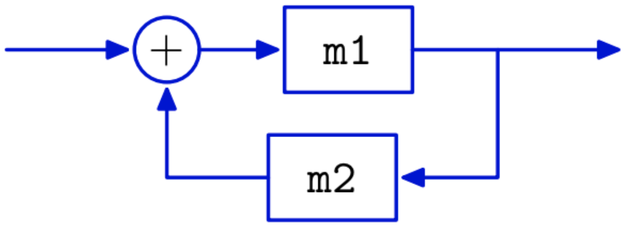
The second part of Figure 4.6 shows a combination we call *feedback2*: it assumes that it takes a machine with two inputs and one output, and connects the output of the machine to the second input, resulting in a machine with one input and one output.

Feedback2 is very similar to the basic feedback combinator, but it gives, as input to the constituent machine, the pair of the input to the machine and the feedback value.

```
class Feedback2 (Feedback):
    def getNextValues(self, state, inp):
        (ignore, o) = self.m.getNextValues(state, (inp, 'undefined'))
        (newS, ignore) = self.m.getNextValues(state, (inp, o))
        return (newS, o)
```

4.2.3.4 FeedbackSubtract and FeedbackAdd

In *feedback addition* composition, we take two machines and connect them as shown below:



If `m1` and `m2` are state machines, then you can create their feedback addition composition with

```
newM = sm.FeedbackAdd(m1, m2)
```

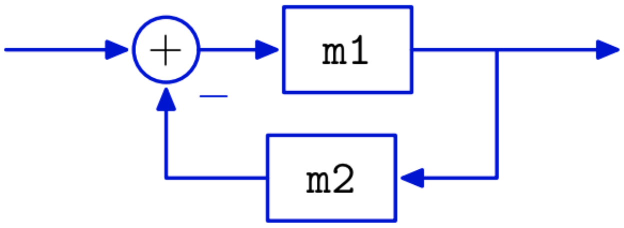
Now `newM` is itself a state machine. So, for example,

```
newM = sm.FeedbackAdd(sm.R(0), sm.Wire())
```

makes a machine whose output is the sum of all the inputs it has ever had (remember that `sm.R` is shorthand for `sm.Delay`). You can test it by feeding it a sequence of inputs; in the example below, it is the numbers 0 through 9:

```
>>> newM.transduce(range(10))
[0, 0, 1, 3, 6, 10, 15, 21, 28, 36]
```

Feedback subtraction composition is the same, except the output of `m2` is *subtracted* from the input, to get the input to `m1`.



Note that if you want to apply one of the feedback operators in a situation where there is only one machine, you can use the `sm.Gain(1.0)` machine, which is essentially a wire, as the other argument.

4.2.3.5 Factorial

We will do one more tricky example, and illustrate the use of `Feedback2`. What if we wanted to generate the sequence of numbers $\{1!, 2!, 3!, 4!, \ldots\}$ (where $k! = 1 \cdot 2 \cdot 3 \ldots \cdot k$)? We can do so by multiplying the previous value of the sequence by a number equal to the "index" of the sequence. Figure 4.9 shows the structure of a machine for solving this problem. It uses a counter (which is, as we saw before, made with feedback around a delay and increment) as the input to a machine that takes a single input, and multiplies it by the output value of the machine, fed back through a delay.

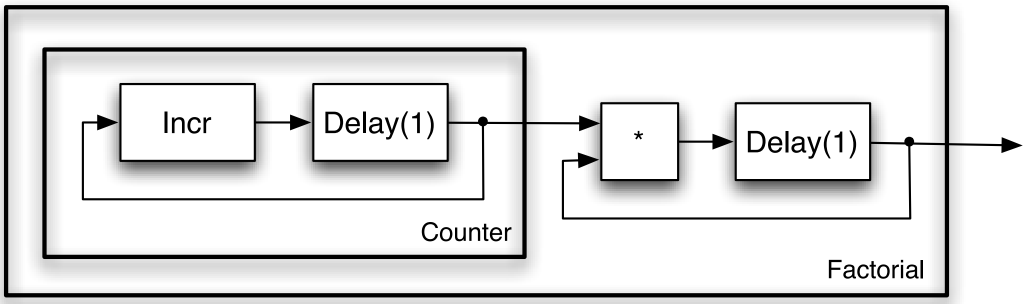


Figure 4.9: Machine to generate the Factorial sequence.

Here is how to do it in Python; we take advantage of having defined counter machines to abstract away from them and use that definition here without thinking about its internal structure. The initial values in the delays get the series started off in the right place. What would happen if we started at 0?

```
fact = sm.Cascade(makeCounter(1, 1),
                  sm.Feedback2(sm.Cascade(Multiplier(), sm.Delay(1))))
```

```
>>> fact.run(verbose = True)
Start state: ((None, 1), (None, 1))
Step: 0
 Cascade_1
     Feedback_2
         Cascade_3
             Increment_4 In: 1 Out: 2 Next State: 2
             Delay_5 In: 2 Out: 1 Next State: 2
     Feedback2_6
         Cascade_7
             Multiplier_8 In: (1, 1) Out: 1 Next State: 1
             Delay_9 In: 1 Out: 1 Next State: 1
Step: 1
 Cascade_1
     Feedback_2
         Cascade_3
             Increment_4 In: 2 Out: 3 Next State: 3
             Delay_5 In: 3 Out: 2 Next State: 3
     Feedback2_6
         Cascade_7
             Multiplier_8 In: (2, 1) Out: 2 Next State: 2
             Delay_9 In: 2 Out: 1 Next State: 2
Step: 2
 Cascade_1
     Feedback_2
         Cascade_3
             Increment_4 In: 3 Out: 4 Next State: 4
             Delay_5 In: 4 Out: 3 Next State: 4
     Feedback2_6
         Cascade_7
             Multiplier_8 In: (3, 2) Out: 6 Next State: 6
             Delay_9 In: 6 Out: 2 Next State: 6
Step: 3
 Cascade_1
     Feedback_2
         Cascade_3
             Increment_4 In: 4 Out: 5 Next State: 5
             Delay_5 In: 5 Out: 4 Next State: 5
     Feedback2_6
         Cascade_7
             Multiplier_8 In: (4, 6) Out: 24 Next State: 24
             Delay_9 In: 24 Out: 6 Next State: 24
...

[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

It might bother you that we get a 1 as the zeroth element of the sequence, but it is reasonable as a definition of 0!, because 1 is the multiplicative identity (and is often defined that way by mathematicians).

4.2.4 Plants and Controllers

One common situation in which we combine machines is to simulate the effects of coupling a controller and a so-called "plant". A plant is a factory or other external environment that we might wish to control. In this case, we connect two state machines so that the output of the plant (typically thought of as sensory observations) is input to the controller, and the output of the controller (typically thought of as actions) is input to the plant. This is shown schematically in Figure 4.10. For example, when you build a Soar brain that interacts with the robot, the robot (and the world in which it is operating) is the "plant" and the brain is the controller. We can build a coupled machine by first connecting the machines in a cascade and then using feedback on that combination.

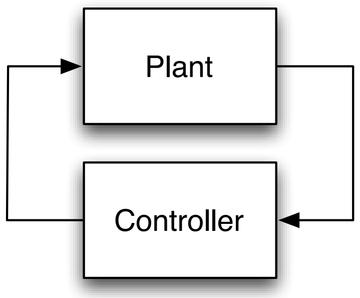


Figure 4.10: Two coupled machines.

As a concrete example, let's think about a robot driving straight toward a wall. It has a distance sensor that allows it to observe the distance to the wall at time t, $d[t]$, and it desires to stop at some distance $d_{desired}$. The robot can execute velocity commands, and we program it to use the following rule to set its velocity at time t, based on its most recent sensor reading:

$$v[t] = K(d_{desired} - d[t-1]) \ .$$

This controller can also be described as a state machine, whose input sequence is the observed values of d and whose output sequence is the values of v.

$$\begin{aligned} S &= numbers \\ I &= numbers \\ O &= numbers \\ n(s,i) &= K(d_{desired} - i) \\ o(s) &= s \\ s_0 &= d_{init} \end{aligned}$$

Now, we can think about the "plant"; that is, the relationship between the robot and the world. The distance of the robot to the wall changes at each time step depending on the robot's forward velocity and the length of the time steps. Let δT be the length of time between velocity commands issued by the robot. Then we can describe the world with the equation:

$$d[t] \;=\; d[t-1] - \delta T v[t-1] \;.$$

which assumes that a positive velocity moves the robot *toward* the wall (and therefore decreases the distance). This system can be described as a state machine, whose input sequence is the values of the robot's velocity, v, and whose output sequence is the values of its distance to the wall, d.

Finally, we can couple these two systems, as for a simulator, to get a single state machine with no inputs. We can observe the sequence of internal values of d and v to understand how the system is behaving.

In Python, we start by defining the controller machine; the values `k` and `dDesired` are constants of the whole system.

```
k = -1.5
dDesired = 1.0
class WallController(SM):
    def getNextState(self, state, inp):
        return safeMul(k, safeAdd(dDesired, safeMul(-1, inp)))
```

The output being generated is actually `k * (dDesired - inp)`, but because this method is going to be used in a feedback machine, it might have to deal with `'undefined'` as an input. It has no delay built into it.

Think about why we want `k` to be negative. What happens when the robot is closer to the wall than desired? What happens when it is farther from the wall than desired?

Now, we can define a class that describes the behavior of the "plant":

```
deltaT = 0.1
class WallWorld(SM):
    startState = 5
    def getNextValues(self, state, inp):
        return (state - deltaT * inp, state)
```

Setting `startState = 5` means that the robot starts 5 meters from the wall. Note that the output of this machine does not depend instantaneously on the input; so there *is* a delay in it.

Now, we can defined a general combinator for coupling two machines, as in a plant and controller:

```
def coupledMachine(m1, m2):
    return sm.Feedback(sm.Cascade(m1, m2))
```

We can use it to connect our controller to the world, and run it:

```
>>> wallSim = coupledMachine(WallController(), WallWorld())
>>> wallSim.run(30)
[5, 4.4000000000000004, 3.8900000000000001, 3.4565000000000001,
 3.088025, 2.77482125, 2.5085980624999999, 2.2823083531249999,
 2.0899621001562498, 1.9264677851328122, 1.7874976173628905,
 1.6693729747584569, 1.5689670285446884, 1.483621974262985,
 1.4110786781235374, 1.3494168764050067, 1.2970043449442556,
 1.2524536932026173, 1.2145856392222247, 1.1823977933388909,
 1.1550381243380574, 1.1317824056873489, 1.1120150448342465,
 1.0952127881091096, 1.0809308698927431, 1.0687912394088317,
 1.058472553497507, 1.049701670472881, 1.0422464199019488,
 1.0359094569166565]
```

Because `WallWorld` is the second machine in the cascade, its output is the output of the whole machine; so, we can see that the distance from the robot to the wall is converging monotonically to `dDesired` (which is 1).

Exercise 4.10
What kind of behavior do you get with different values of `k`?

4.2.5 Conditionals

We might want to use different machines depending on something that is happening in the outside world. Here we describe three different conditional combinators, that make choices, at the run-time of the machine, about what to do.

4.2.6 Switch

We will start by considering a conditional combinator that runs two machines in parallel, but decides *on every input* whether to send the input into one machine or the other. So, only one of the parallel machines has its state updated on each step. We will call this *switch*, to emphasize the fact that the decision about which machine to execute is being re-made on every step.

Implementing this requires us to maintain the states of both machines, just as we did for parallel combination. The `getNextValues` method tests the condition and then gets a new state and output from the appropriate constituent machine; it also has to be sure to pass through the old state for the constituent machine that was not updated this time.

```
class Switch (SM):
    def __init__(self, condition, sm1, sm2):
        self.m1 = sm1
        self.m2 = sm2
        self.condition = condition
        self.startState = (self.m1.startState, self.m2.startState)

    def getNextValues(self, state, inp):
        (s1, s2) = state
        if self.condition(inp):
            (ns1, o) = self.m1.getNextValues(s1, inp)
            return ((ns1, s2), o)
        else:
            (ns2, o) = self.m2.getNextValues(s2, inp)
            return ((s1, ns2), o)
```

4.2.7 Multiplex

The `switch` combinator takes care to only update one of the component machines; in some other cases, we want to update both machines on every step and simply use the condition to select the output of one machine or the other to be the current output of the combined machine.

This is a very small variation on `Switch`, so we will just implement it as a subclass.

```
class Mux (Switch):
    def getNextValues(self, state, inp):
        (s1, s2) = state
        (ns1, o1) = self.m1.getNextValues(s1, inp)
        (ns2, o2) = self.m2.getNextValues(s2, inp)
        if self.condition(inp):
            return ((ns1, ns2), o1)
        else:
            return ((ns1, ns2), o2)
```

Exercise 4.11
What is the result of running these two machines

```
m1 = Switch(lambda inp: inp > 100,
            Accumulator(),
            Accumulator())
m2 = Mux(lambda inp: inp > 100,
         Accumulator(),
         Accumulator())
```

on the input

```
[2, 3, 4, 200, 300, 400, 1, 2, 3]
```

Explain why they are the same or are different.

4.3 Terminating State Machines and Sequential Compositions

So far, all the machines we have discussed run forever; or, at least, until we quit giving them inputs. But in some cases, it is particularly useful to think of a process as consisting of a sequence of processes, one executing until termination, and then another one starting. For example, you might want to robot to clean first room A, **and then** clean room B; or, for it to search in an area **until** it finds a person **and then** sound an alarm.

Temporal combinations of machines form a new, different PCAP system for state machines. Our primitives will be state machines, as described above, but with one additional property: they will have a termination or *done* function, $d(s)$, which takes a state and returns *true* if the machine has finished execution and *false* otherwise.

Rather than defining a whole new class of state machines (though we could do that), we will just augment the `SM` class with a default method, which says that, by default, machines do not terminate.

```
    def done(self, state):
        return False
```

Then, in the definition of any subclass of `SM`, you are free to implement your own `done` method that will override this base one. The `done` method is used by state machine combinators that, for example, run one machine until it is done, and then switch to running another one.

Here is an example *terminating state machine* (TSM) that consumes a stream of numbers; its output is `None` on the first four steps and then on the fifth step, it generates the sum of the numbers it has seen as inputs, and then terminates. It looks just like the state machines we have seen before, with the addition of a `done` method. Its state consists of two numbers: the first is the number of times the machine has been updated and the second is the total input it has accumulated so far.

```
class ConsumeFiveValues(SM):
    startState = (0, 0)      # count, total

    def getNextValues(self, state, inp):
        (count, total) = state
        if count == 4:
            return ((count + 1, total + inp), total + inp)
        else:
            return ((count + 1, total + inp), None)

    def done(self, state):
        (count, total) = state
        return count == 5
```

Here is the result of running a simple example. We have modified the `transduce` method of `SM` to stop when the machine is done.

```
>>> c5 = ConsumeFiveValues()
>>> c5.transduce([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], verbose = True)
Start state: (0, 0)
In: 1 Out: None Next State: (1, 1)
In: 2 Out: None Next State: (2, 3)
In: 3 Out: None Next State: (3, 6)
In: 4 Out: None Next State: (4, 10)
In: 5 Out: 15 Next State: (5, 15)
[None, None, None, None, 15]
```

Now we can define a new set of combinators that operate on TSMs. Each of these combinators assumes that its constituent machines are terminating state machines, and are, themselves, terminating state machines. We have to respect certain rules about TSMs when we do this. In particular, it is not legal to call the `getNextValues` method on a TSM that says it is done. This may or may not cause an actual Python error, but it is never a sensible thing to do, and may result in meaningless answers.

4.3.1 Repeat

The simplest of the TSM combinators is one that takes a terminating state machine `sm` and repeats it `n` times. In the Python method below, we give a default value of `None` for `n`, so that if no value is passed in for `n` it will repeat forever.

```
class Repeat (SM):
    def __init__(self, sm, n = None):
        self.sm = sm
        self.startState = (0, self.sm.startState)
        self.n = n
```

The state of this machine will be the number of times the constituent machine has been executed to completion, together with the current state of the constituent machine. So, the starting state is a pair consisting of `0` and the starting state of the constituent machine.

Because we are going to, later, ask the constituent machine to generate an output, we are going to adopt a convention that the constituent machine is never left in a state that is done, unless the whole `Repeat` is itself done. If the constituent machine is done, then we will increment the counter for the number of times we have repeated it, and restart it. Just in case the constituent machine "wakes up" in a state that is done, we use a `while` loop here, instead of an `if`: we will keep restarting this machine until the count runs out. Why? Because we promised not to leave our constituent machine in a done state (so, for example, nobody asks for its output, when its done), unless the whole `repeat` machine is done as well.

```
    def advanceIfDone(self, counter, smState):
        while self.sm.done(smState) and not self.done((counter, smState)):
            counter = counter + 1
            smState = self.sm.startState
        return (counter, smState)
```

To get the next state, we start by getting the next state of the constituent machine; then, we check to see if the counter needs to be advanced and the machine restarted; the `advanceIfDone` method handles this situation and returns the appropriate next state. The output of the `Repeat` machine is just the output of the constituent machine. We just have to be sure to destructure the state of the overall machine and pass the right part of it into the constituent.

```
    def getNextValues(self, state, inp):
        (counter, smState) = state
        (smState, o) = self.sm.getNextValues(smState, inp)
        (counter, smState) = self.advanceIfDone(counter, smState)
        return ((counter, smState), o)
```

We know the whole `Repeat` is done if the counter is equal to `n`.

```
    def done(self, state):
        (counter, smState) = state
        return counter == self.n
```

Now, we can see some examples of `Repeat`. As a primitive, here is a silly little example TSM. It takes a character at initialization time. Its state is a Boolean, indicating whether it is done. It starts up in state `False` (not done). Then it makes its first transition into state `True` and stays there. Its output is always the character it was initialized with; it completely ignores its input.

```
class CharTSM (SM):
    startState = False
    def __init__(self, c):
        self.c = c
    def getNextValues(self, state, inp):
        return (True, self.c)
    def done(self, state):
        return state

>>> a = CharTSM('a')
>>> a.run(verbose = True)
Start state: False
In: None Out: a Next State: True
['a']
```

See that it terminates after one output. But, now, we can repeat it several times.

```
>>> a4 = sm.Repeat(a, 4)
>>> a4.run()
['a', 'a', 'a', 'a']
```

Exercise 4.12
Would it have made a difference if we had executed:

```
>>> sm.Repeat(CharTSM('a'), 4).run()
```

Exercise 4.13
Monty P. thinks that the following call

```
>>> sm.Repeat(ConsumeFiveValues(), 3).transduce(range(100))
```

will generate a sequence of 14 `None`s followed by the sum of the first 15 integers (starting at 0). R. Reticulatis disagrees. Who is right and why?

4.3.2 Sequence

Another useful thing to do with TSMs is to execute several different machines sequentially. That is, take a list of TSMs, run the first one until it is done, start the next one and run it until it is done, and so on. This machine is similar in style and structure to a `Repeat` TSM. Its state is a pair of values: an index that says which of the constituent machines is currently being executed, and the state of the current constituent.

Here is a Python class for creating a `Sequence` TSM. It takes as input a list of state machines; it remembers the machines and number of machines in the list.

```
class Sequence (SM):
    def __init__(self, smList):
        self.smList = smList
        self.startState = (0, self.smList[0].startState)
        self.n = len(smList)
```

The initial state of this machine is the value 0 (because we start by executing the 0th constituent machine on the list) and the initial state of that constituent machine.

The method for advancing is also similar that for `Repeat`. The only difference is that each time, we start the next machine in the list of machines, until we have finished executing the last one.

```
    def advanceIfDone(self, counter, smState):
        while self.smList[counter].done(smState) and counter + 1 < self.n:
            counter = counter + 1
            smState = self.smList[counter].startState
        return (counter, smState)
```

To get the next state, we ask the current constituent machine for its next state, and then, if it is done, advance the state to the next machine in the list that is not done when it wakes up. The output of the composite machine is just the output of the current constituent.

```
    def getNextValues(self, state, inp):
        (counter, smState) = state
        (smState, o) = self.smList[counter].getNextValues(smState, inp)
        (counter, smState) = self.advanceIfDone(counter, smState)
        return ((counter, smState), o)
```

We have constructed this machine so that it always advances past any constituent machine that is done; if, in fact, the current constituent machine is done, then the whole machine is also done.

```
    def done(self, state):
        (counter, smState) = state
        return self.smList[counter].done(smState)
```

We can make good use of the `CharTSM` to test our sequential combinator. First, we will try something simple:

```
>>> m = sm.Sequence([CharTSM('a'), CharTSM('b'), CharTSM('c')])
>>> m.run()
Start state: (0, False)
In: None Out: a Next State: (1, False)
In: None Out: b Next State: (2, False)
In: None Out: c Next State: (2, True)
['a', 'b', 'c']
```

Even in a test case, there is something unsatisfying about all that repetitive typing required to make each individual `CharTSM`. If we are repeating, we should abstract. So, we can write a function that takes a string as input, and returns a sequential TSM that will output that string. It uses a list comprehension to turn each character into a `CharTSM` that generates that character, and then uses that sequence to make a `Sequence`.

```
def makeTextSequenceTSM(str):
    return sm.Sequence([CharTSM(c) for c in str])

>>> m = makeTextSequenceTSM('Hello World')
>>> m.run(20, verbose = True)
Start state: (0, False)
In: None Out: H Next State: (1, False)
In: None Out: e Next State: (2, False)
In: None Out: l Next State: (3, False)
In: None Out: l Next State: (4, False)
In: None Out: o Next State: (5, False)
In: None Out:   Next State: (6, False)
In: None Out: W Next State: (7, False)
In: None Out: o Next State: (8, False)
In: None Out: r Next State: (9, False)
In: None Out: l Next State: (10, False)
In: None Out: d Next State: (10, True)
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

We can also see that sequencing interacts well with the `Repeat` combinator.

```
>>> m = sm.Repeat(makeTextSequenceTSM('abc'), 3)
>>> m.run(verbose = True)
Start state: (0, (0, False))
In: None Out: a Next State: (0, (1, False))
In: None Out: b Next State: (0, (2, False))
In: None Out: c Next State: (1, (0, False))
In: None Out: a Next State: (1, (1, False))
In: None Out: b Next State: (1, (2, False))
In: None Out: c Next State: (2, (0, False))
In: None Out: a Next State: (2, (1, False))
In: None Out: b Next State: (2, (2, False))
In: None Out: c Next State: (3, (0, False))
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

It is interesting to understand the state here. The first value is the number of times the constituent machine of the `Repeat` machine has finished executing; the second value is the index of the sequential machine into its list of machines, and the last Boolean is the state of the `CharTSM` that is being executed, which is an indicator for whether it is done or not.

4.4 Using a State Machine to Control the Robot

This section gives an overview of how to control the robot with a state machine. For a much more detailed description, see the *Infrastructure Guide*, which documents the `io` and `util` modules in detail. The `io` module provides procedures and methods for interacting with the robot; the `util` module provides procedures and methods for doing computations that are generally useful (manipulating angles, dealing with coordinate frames, etc.)

We can implement a robot controller as a state machine whose inputs are instances of class `io.SensorInput`, and whose outputs are instances of class `io.Action`.

Here is Python code for a brain that is controlled by the most basic of state machines. This machine always emits the default action, `io.Action()`, which sets all of the output values to zero. When the brain is set up, we create a "behavior", which is a name we will use for a state machine that transduces a stream of `io.SensorInputs` to a stream of `io.Actions`. Finally, we ask the behavior to start.

Then, all we do in the `step` method of the robot is:

- Read the sensors, by calling `io.SensorInput()` to get an instance that contains sonar and odometry readings;
- Feed that sensor input to the brain state machine, by calling its `step` method with that as input; and
- Take the `io.Action` that is generated by the brain as output, and call its `execute` method, which causes it to actually send motor commands to the robot.

You can set the `verbose` flag to `True` if you want to see a lot of output on each step for debugging.

Inside a Soar brain, we have access to an object `robot`, which persists during the entire execution of the brain, and gives us a place to store important objects (like the state machine that will be doing all the work).

```
import sm
import io

class StopSM(sm.SM):
    def getNextValues(self, state, inp):
        return (None, io.Action())

def setup():
    robot.behavior = StopSM()
    robot.behavior.start()
```

```
def step():
    robot.behavior.step(io.SensorInput(), verbose = False).execute()
```

In the following sections we will develop two simple machines for controlling a robot to move a fixed distance or turn through a fixed angle. Then we will put them together and explore why it can be useful to have the starting state of a machine depend on the input.

4.4.1 Rotate

Imagine that we want the robot to rotate a fixed angle, say 90 degrees, to the left of where it is when it starts to run a behavior. We can use the robot's odometry to measure approximately where it is, in an arbitrary coordinate frame; but to know how much it has moved since we started, we have to store some information in the state.

Here is a class that defines a `Rotate` state machine. It takes, at initialization time, a desired change in heading.

```
class RotateTSM (SM):
    rotationalGain = 3.0
    angleEpsilon = 0.01
    startState = 'start'

    def __init__(self, headingDelta):
        self.headingDelta = headingDelta
```

When it is time to start this machine, we would like to look at the robot's current heading (`theta`), add the desired change in heading, and store the result in our state as the desired heading. Then, in order to test whether the behavior is done, we want to see whether the current heading is close enough to the desired heading. Because the `done` method does not have access to the input of the machine (it is a property only of states), we need to include the current `theta` in the state. So, the state of the machine is `(thetaDesired, thetaLast)`.

Thus, the `getNextValues` method looks at the state; if it is the special symbol `'start'`, it means that the machine has not previously had a chance to observe the input and see what its current heading is, so it computes the desired heading (by adding the desired change to the current heading, and then calling a utility procedure to be sure the resulting angle is between plus and minus π), and returns it and the current heading. Otherwise, we keep the `thetaDesired` component of the state, and just get a new value of `theta` out of the input. We generate an action with a rotational velocity that will rotate toward the desired heading with velocity proportional to the magnitude of the angular error.

```
    def getNextValues(self, state, inp):
        currentTheta = inp.odometry.theta
        if state == 'start':
            thetaDesired = \
               util.fixAnglePlusMinusPi(currentTheta + self.headingDelta)
        else:
            (thetaDesired, thetaLast) = state
        newState = (thetaDesired, currentTheta)
        action = io.Action(rvel = self.rotationalGain * \
                     util.fixAnglePlusMinusPi(thetaDesired - currentTheta))

        return (newState, action)
```

Finally, we have to say which states are done. Clearly, the `'start'` state is not done; but we are done if the most recent `theta` from the odometry is within some tolerance, `self.angleEpsilon`, of the desired heading.

```
    def done(self, state):
        if state == 'start':
            return False
        else:
            (thetaDesired, thetaLast) = state
            return util.nearAngle(thetaDesired, thetaLast, self.angleEpsilon)
```

Exercise 4.14
Change this machine so that it rotates *through* an angle, so you could give it 2 pi or minus 2 pi to have it rotate all the way around.

4.4.2 Forward

Moving the robot forward a fixed distance is similar. In this case, we remember the robot's `x` and `y` coordinates when it starts, and drive straight forward until the distance between the initial position and the current position is close to the desired distance. The state of the machine is the robot's starting position and its current position.

```
class ForwardTSM (SM):
    forwardGain = 1.0
    distTargetEpsilon = 0.01
    startState = 'start'

    def __init__(self, delta):
        self.deltaDesired = delta

    def getNextValues(self, state, inp):
        currentPos = inp.odometry.point()
        if state == 'start':
            print "Starting forward", self.deltaDesired
            startPos = currentPos
        else:
            (startPos, lastPos) = state
        newState = (startPos, currentPos)
        error = self.deltaDesired - startPos.distance(currentPos)
        action = io.Action(fvel = self.forwardGain * error)
        return (newState, action)

    def done(self, state):
        if state == 'start':
            return False
        else:
            (startPos, lastPos) = state
            return util.within(startPos.distance(lastPos),
                               self.deltaDesired,
                               self.distTargetEpsilon)
```

4.4.3 Square Spiral

Imagine we would like to have the robot drive in a square spiral, similar to the one shown in Figure 4.11. One way to approach this problem is to make a "low-level" machine that can consume a goal point and the sensor input and drive (in the absence of obstacles) to the goal point; and then make a "high-level" machine that will keep track of where we are in the figure and feed goal points to the low-level machine.

4.4.3.1 XYDriver

Here is a class that describes a machine that takes as input a series of pairs of goal points (expressed in the robot's odometry frame) and sensor input structures. It generates as output a series of actions. This machine is very nearly a pure function machine, which has the following basic control structure:

- If the robot is headed toward the goal point, move forward.
- If it is not headed toward the goal point, rotate toward the goal point.

This decision is made on every step, and results in a robust ability to drive toward a point in two-dimensional space.

For many uses, this machine does not need any state. But the modularity is nicer, in some cases, if it has a meaningful `done` method, which depends only on the state. So, we will let the state of this machine be whether it is done or not. It needs several constants

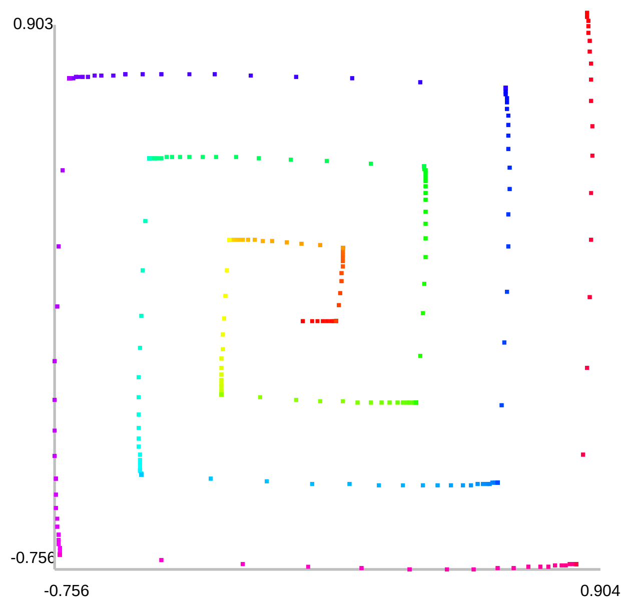


Figure 4.11: Square spiral path of the robot using the methods in this section.

to govern rotational and forward speeds, and tolerances for deciding whether it is pointed close enough toward the target and whether it has arrived close enough to the target.

```
class XYDriver(SM):
    forwardGain = 2.0
    rotationGain = 2.0
    angleEps = 0.05
    distEps = 0.02
    startState = False
```

The `getNextValues` method embodies the control structure described above.

```
    def getNextValues(self, state, inp):
        (goalPoint, sensors) = inp
        robotPose = sensors.odometry
        robotPoint = robotPose.point()
        robotTheta = robotPose.theta

        if goalPoint == None:
            return (True, io.Action())

        headingTheta = robotPoint.angleTo(goalPoint)
        if util.nearAngle(robotTheta, headingTheta, self.angleEps):
            # Pointing in the right direction, so move forward
            r = robotPoint.distance(goalPoint)
            if r < self.distEps:
                # We're there
                return (True, io.Action())
            else:
                return (False, io.Action(fvel = r * self.forwardGain))
        else:
            # Rotate to point toward goal
            headingError = util.fixAnglePlusMinusPi(\
                                                 headingTheta - robotTheta)
            return (False, io.Action(rvel = headingError * self.rotationGain))
```

The state of the machine is just a boolean indicating whether we are done.

```
    def done(self, state):
        return state
```

4.5 Conclusion

State machines are such a general formalism, that a huge class of discrete-time systems can be described as state machines. The system of defining primitive machines and combinations gives us one discipline for describing complex systems. It will turn out that there are some systems that are conveniently defined using this discipline, but that for other kinds of systems, other disciplines would be more natural. As you encounter complex engineering problems, your job is to find the PCAP system that is appropriate for them, and if one does not exist already, invent one.

State machines are such a general class of systems that although it is a useful framework for implementing systems, we cannot generally analyze the behavior of state machines. That is, we can't make much in the way of generic predictions about their future behavior, except by running them to see what will happen.

In the next module, we will look at a restricted class of statef machines, whose state is representable as a bounded history of their previous states and previous inputs, and whose output is a linear function of those states and inputs. This is a *much* smaller class of systems than all state machines, but it is nonetheless very powerful. The important lesson will be that restricting the form of the models we are using will allow us to make stronger claims about their behavior.

Knuth on Elevator Controllers

Donald E. Knuth is a computer scientist who is famous for, among other things, his series of textbooks (as well as for TEX, the typesetting system we use to make all of our handouts), and a variety of other contributions to theoretical computer science.

"It is perhaps significant to note that although the author had used the elevator system for years and thought he knew it well, it wasn't until he attempted to write this section that he realized there were quite a few facts about the elevator's system of choosing directions that he did not know. He went back to experiment with the elevator six separate times, each time believing he had finally achieved a complete understanding of its *modus operandi.* (Now he is reluctant to ride it for fear some new facet of its operation will appear, contradicting the algorithms given.) We often fail to realize how little we know about a thing until we attempt to simulate it on a computer."

The Art of Computer Programming, Donald E., Knuth, Vol 1. page 295. On the elevator system in the Mathematics Building at Cal Tech. First published in 1968

4.6 Examples

4.6.1 Practice Problem: Things

Consider the following program

```
def thing(inputList):
    output = []
    i = 0
    for x in range(3):
        y = 0
        while y < 100 and i < len(inputList):
            y = y + inputList[i]
            output.append(y)
            i = i + 1
    return output
```

A. What is the value of

```
thing([1, 2, 3, 100, 4, 9, 500, 51, -2, 57, 103, 1, 1, 1, 1, -10, 207, 3, 1])
```

```
Ans: [1, 3, 6, 106, 4, 13, 513, 51, 49, 106]
```

It's important to understand the loop structure of the Python program: It goes through (at most) three times, and adds up the elements of the input list, generating a partial sum as output on each step, and terminating the inner loop when the sum becomes greater than 100.

B. Write a single state machine class `MySM` such that `MySM().transduce(inputList)` gives the same result as `thing(inputList)`, if `inputList` is a list of numbers. Remember to include a `done` method, that will cause it to terminate at the same time as `thing`.

Ans:

```
class MySM(sm.SM):
    startState = (0,0)
    def getNextValues(self, state, inp):
        (x, y) = state
        y += inp
        if y >= 100:
            return ((x + 1, 0), y)
        return ((x, y), y)
    def done(self, state):
        (x, y) = state
        return x >= 3
```

The most important step, conceptually, is deciding what the state of the machine will be. Looking at the original Python program, we can see that we had to keep track of how many times we had completed the outer loop, and then what the current partial sum was of the inner loop.

The `getNextValues` method first increments the partial sum by the input value, and then checks to see whether it's time to reset. If so, it increments the 'loop counter' (`x`) component of the state and resets the partial sum to 0. It's important to remember that the output of the `getNextValues` method is a pair, containing the next state and the output.

The `done` method just checks to see whether we have finished three whole iterations.

C. Recall the definition of `sm.Repeat(m, n)`: Given a terminating state machine `m`, it returns a new terminating state machine that will execute the machine `m` to completion `n` times, and then terminate.

Use `sm.Repeat` and a very simple state machine that you define to create a new state machine `MyNewSM`, such that `MyNewSM` is equivalent to an instance of `MySM`.

Ans:

```
class Sum(sm.SM):
    startState = 0
    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
    def done(self, state):
        return state > 100
```

```
myNewSM = sm.Repeat(Sum(), 3)
```

Chapter 5

Signals and Systems

Imagine that you are asked to design a system to steer a car straight down the middle of a lane. It seems easy, right? You can figure out some way to sense the position of the car within its lane. Then, if the car is right of center, turn the steering wheel to the left. As the car moves so that it is less to the right, turn the steering wheel less to the left. If it is left of center, turn the steering wheel to the right. This sort of *proportional controller* works well for many applications – but not for steering, as can be seen below.

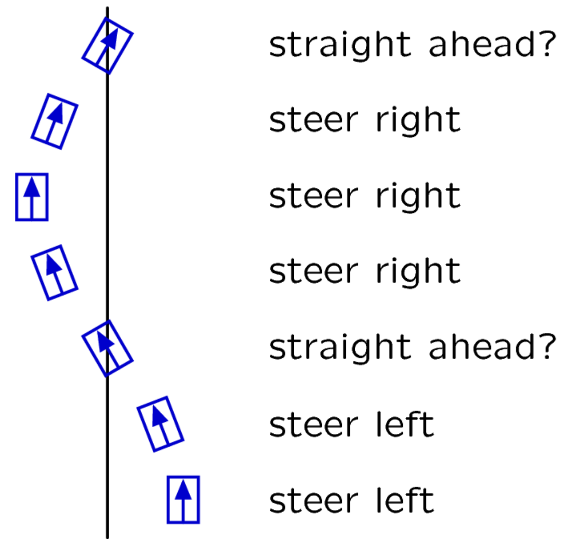


Figure 5.1: Simple (but poor) algorithm for steering: steer to the left in proportion to how far the car is to the right, and vice versa.

It is relatively easy to describe better algorithms in terms that humans would understand: e.g., *Stop turning back and forth!* It is not so easy to specify exactly what one might mean by that, in a way that it could be automated.

In this chapter, we will develop a *Signals and Systems* framework to facilitate reasoning about the dynamic behaviors of systems. This framework will enable construction of simple mathematical models that are useful in both analysis and design of a wide range of systems, including the car-steering system.

5.1 The Signals and Systems Abstraction

To think about dynamic behaviors of systems, we need to think not only about how to describe the system but also about how to describe the *signals* that characterize the inputs and outputs of the system, as illustrated below.

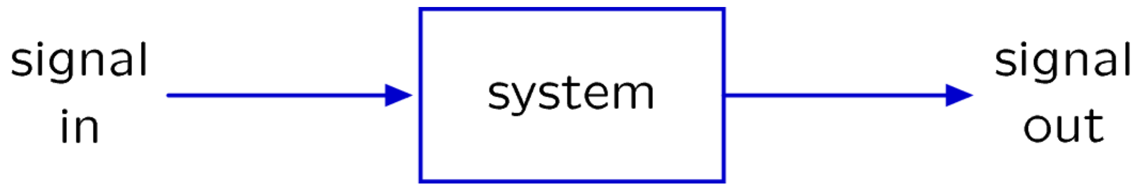


Figure 5.2: Signals and Systems: the system transforms an input signal into an output signal.

This diagram represents a system with one input and one output. Both the input and output are *signals*. A signal is a mathematical function with an independent variable (most often it will be *time* for the problems that we will study) and a dependent variable (that depends on the independent variable). The *system* is described by the way that it transforms the input signal into the output signal. In the simplest case, we might imagine that the input signal is the time sequence of steering-wheel angles (assuming constant speed) and that the output signal is the time sequence of distances between the center of the car and the midline of the lane.

Representing a system with a single input signal and a single output signal seems too simplistic for any real application. For example, the car in the steering example (Figure 5.1) surely has more than one possible output signal.

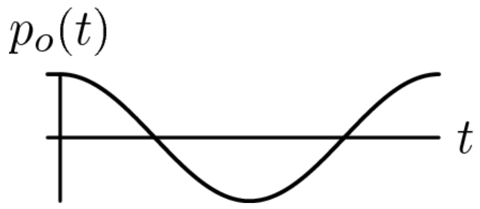
Exercise 5.1
List at least four possible output signals for the car-steering problem.

Possible output signals include

- its three-dimensional position (which could be represented by a 3D vector $\hat{p}(t)$ or by three scalar functions of time),
- its angular position,
- the rotational speeds of the wheels,
- the temperature of the tires, and many other possibilities.

The important point is that the first step in using the signals and systems representation is *abstraction*: we must choose the outputs that are most relevant to the problem at hand and abstract away the rest.

To understand the steering of a car, one vital output signal is the lateral position $p_o(t)$ within the lane, where $p_o(t)$ represents the distance (in meters) from the center of the lane. That signal alone tells us a great deal about how well we are steering. Consider a plot of $p_o(t)$ that corresponds to Figure 5.1, as follows.



The oscillations in $p_o(t)$ as a function of time correspond to the oscillations of the car within its lane. Thus, this signal clearly represents an important failure mode of our car steering system.

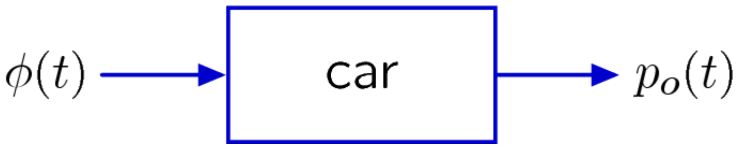
Is $p_o(t)$ the only important output signal from the car-steering system? The answer to this question depends on your goals. Analyzing a system with this single output is likely to give important insights into some systems (e.g., low-speed robotic steering) but not others (e.g., NASCAR). More complicated applications may require more complicated models. But all useful models focus on the most relevant signals and ignore those of lesser significance.[1]

Throughout this chapter, we will focus on systems with one input signal and one output signal (as illustrated Figure 5.2). When multiple output signals are important for understanding a problem, we will find that it is possible to generalize the methods and results developed here for single-input and single-output systems to systems with multiple inputs and outputs.

The signals and systems approach has very broad applicability: it can be applied to mechanical systems (such as mass-spring systems), electrical systems (such as circuits and radio transmissions), financial systems (such as markets), architectural systems (such as heating, ventilation, and air-conditioning (HVAC) systems), and biological systems (such as insulin regulation or population dynamics). The fundamental notion of signals applies no matter what physical substrate supports them: it could be sound or electromagnetic waves or light or water or currency value or blood sugar levels.

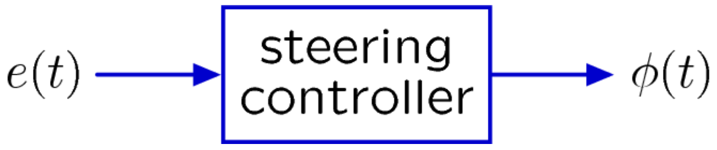
5.1.1 Modularity, Primitives, and Composition

The car-steering system can be analyzed by thinking of it as the combination of car and steering sub-systems. The input to the car is the angle of the steering wheel. Let's call that angle $\phi(t)$. The output of the car is its position in the lane, $p_o(t)$, measured as the lateral distance to the center of the lane.



[1] There are always unimportant outputs. Think about the number of moving parts in a car. They are not all important for steering!

The steering controller turns the steering wheel to compensate for differences between our desired position in the lane, $p_i(t)$ (which is zero since we would like to be in the center of the lane), and our actual position in the lane $p_o(t)$. Let $e(t) = p_i(t) - p_o(t)$. Thus we can think about the steering controller as having an input $e(t)$ and output $\phi(t)$.



In the composite system (Figure 5.3), the steering controller determines $\phi(t)$, which is the input to the car. The car generates $p_o(t)$, which is subtracted from $p_i(t)$ to get $e(t)$ (which is the input to the steering controller). The triangular component is called a *gain* or *scale* of -1: its output is equal to -1 times its input. More generally, we will use a triangle symbol to indicate that we are multiplying all the values of the signal by a numerical constant, which is shown inside the triangle.

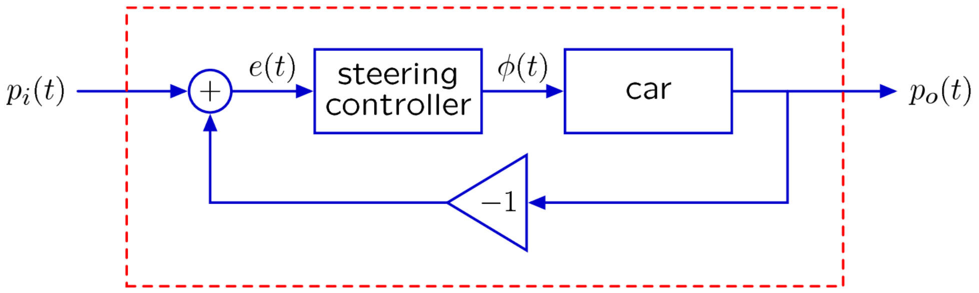


Figure 5.3: Modularity of systems

The dashed-red box in Figure 5.3 illustrates *modularity* of the signals and systems abstraction. Three single-input, single-output sub-systems (steering controller, car, and inverter) and an adder (two inputs and 1 output) are combined to generate a new single-input ($p_i(t)$), single-output ($p_o(t)$) system. By abstraction, we could treat this new system as a primitive (represented by a single-input single-output box) and combine it with other subsystems to create a new, more complex, system. A principal goal of this chapter is to develop methods of analysis for the sub-systems that can be *combined* to analyze the overall system.

5.1.2 Discrete-Time Signals and Systems

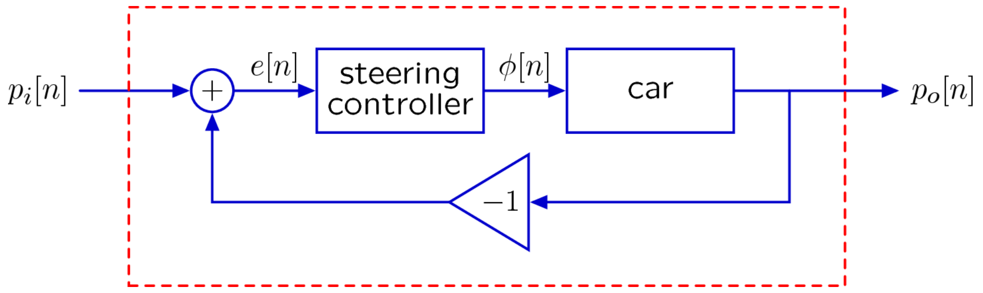
This chapter focuses on signals whose independent variables are discrete (e.g., take on only integer values). Some such signals are found in nature. For example, the primary structure of DNA is described by a *sequence* of base-pairs. However, we are primarily interested in discrete-time signals, not so much because they are found in nature, but because they are found in computers. Even though we focus on interactions with the real world, these interactions will primarily occur at discrete instants of time. For example, the difference between our desired position $p_i(t)$ and our actual position $p_o(t)$ is an error signal $e(t)$, which is a function of continuous time t. If the controller only observes this signal at regular sampling intervals T, then its input could be regarded as a sequence of values $x[n]$ that is indexed by the integer n. The relation between the discrete-time sequence $x[n]$ (note square brackets) and the continuous signal $x(t)$ (note round brackets) is given by

$$x[n] = x(nT) \ ,$$

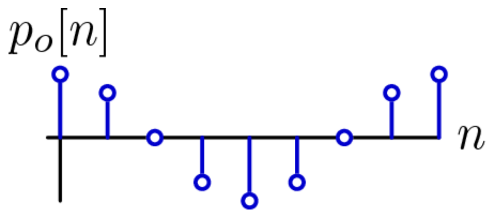
which we call the *sampling* relation. *Sampling* converts a signal of continuous domain to one of discrete domain.

While our primary focus will be on time signals, sampling works equally well in other domains. For example, images are typically represented as arrays of pixels accessed by integer-valued rows and columns, rather than as continuous brightness fields, indexed by real-valued spatial coordinates.

If the car-steering problem in Figure 5.1 were modeled in discrete time, we could describe the system with a diagram that is very similar to the continuous-time diagram in Figure 5.3. However, only discrete time instants are considered



and the output position is now only defined at discrete times, as shown below.



5.1.3 Linear Time-Invariant Systems

We already have a great way of specifying systems that operate on discrete-time signals: a state machine transduces a discrete-time input signal into a discrete-time output signal. State machines, as we have defined them, allow us to specify *any* discrete-time system whose output is computable from its history of previous inputs.

The representation of systems as state machines allows us to execute a machine on any input we'd like, in order to see what happens. Execution lets us examine the behavior of the system for any particular input for any particular finite amount of time, but it does not let us characterize any general properties of the system or its long-term behavior.

Computer programs are such a powerful specification language that we cannot, in general, predict what a program will do (or even whether it will ever stop and return a value) without running it. In the rest of this chapter, we will concentrate on a small but powerful subclass of the whole class of state machines, called discrete-time *linear time-invariant (LTI) systems*, which will allow deeper forms of analysis.

In an LTI system:

- Inputs and outputs are real numbers;
- The state is some fixed number of previous inputs to the system as well as a fixed number of previous outputs of the system; and
- The output is a fixed, linear function of the current input and any of the elements of the state.

In general, each input could be a fixed-length vector of numbers, and each output could also be a fixed-length vector of numbers; we will restrict our attention to the case where the input is a single real number and the output is a single real number.

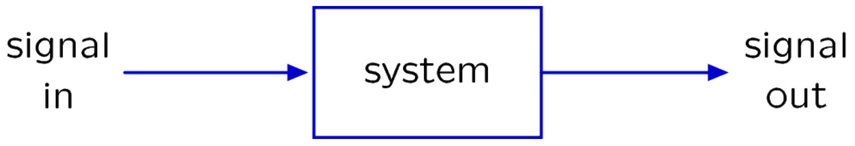
We are particularly interested in LTI systems because they can be analyzed mathematically, in a way that lets us characterize some properties of their output signal for *any* possible input signal. This is a much more powerful kind of insight than can be gained by trying a machine out with several different inputs.

Another important property of LTI systems is that they are compositional: the cascade, parallel, and feedback combinations (introduced in Section 4.2) of LTI system are themselves LTI systems.

5.2 Discrete-Time Signals

In this section, we will work through the PCAP system for discrete time signals, by introducing a primitive and three methods of composition, and the ability to abstract by treating composite signals as if they themselves were primitive.

A *signal* is an infinite sequence of *sample* values at discrete time steps. We will use the following common notational conventions: A capital letter X stands for the whole input signal and $x[n]$ stands for the value of signal X at time step n. It is conventional, if there is a single system under discussion, to use X for the input signal to that system and Y for the output signal.



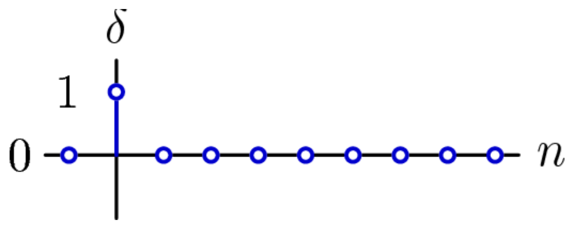
We will say that systems *transduce* input signals into output signals.

5.2.1 Unit Sample Signal

We will work with a single primitive, called the *unit sample signal*, Δ. It is defined on all positive and negative integer indices as follows[2]:

$$\delta[n] = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases}$$

That is, it has value 1 at index $n = 0$ and 0 otherwise, as shown below:

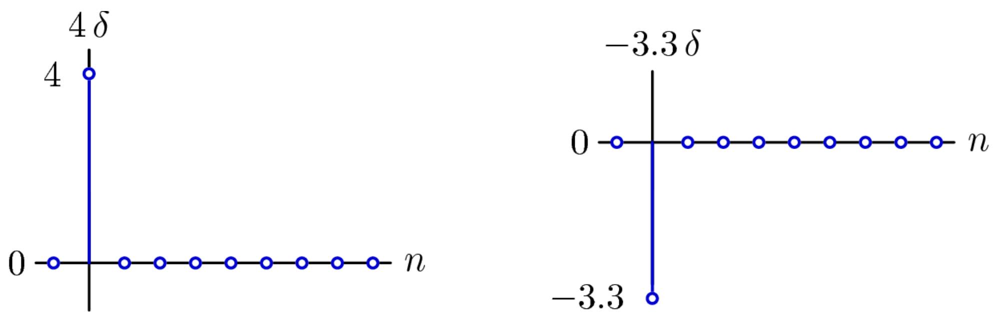


5.2.2 Signal Combinators

Our first operation will be *scaling*, or *multiplication by a scalar*. A scalar is any real number. The result of multiplying any signal X by a scalar c is a signal, so that,

$$\text{if } \; Y = c \cdot X \; \text{ then } \; y[n] = c \cdot x[n] \; .$$

That is, the resulting signal has a value at every index n that is c times the value of the original signal at that location. Here are the signals 4Δ and -3.3Δ.

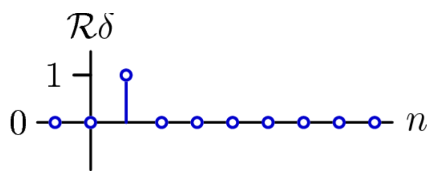


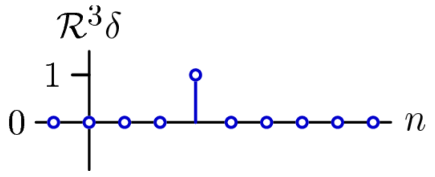
The next operation is the *delay* operation. The result of delaying a signal X is a new signal $\mathcal{R}X$ such that:

$$\text{if } \; Y = \mathcal{R}X \; \text{ then } \; y[n] = x[n-1] \; .$$

That is, the resulting signal has the same values as the original signal, but delayed by one step in time. You can also think of this, graphically, as shifting the signal one step to the $\mathcal{R}$ight. Here is the unit sample delayed by 1 and by 3 steps. We can describe the second signal as $\mathcal{RRR}\Delta$, or, using shorthand, as $\mathcal{R}^3\Delta$.

[2]Note that δ is the lowercase version of Δ, both of which are the Greek letter 'delta'.

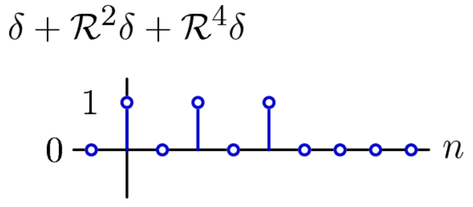


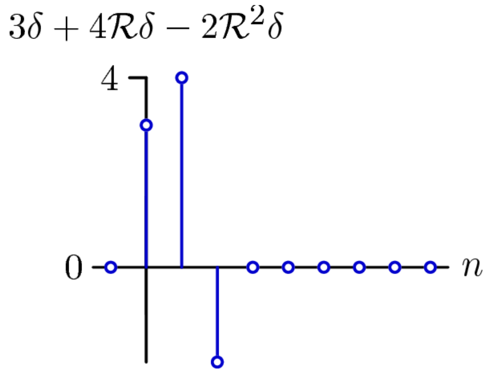


Finally, we can *add* two signals together. Addition of signals is accomplished componentwise, so that

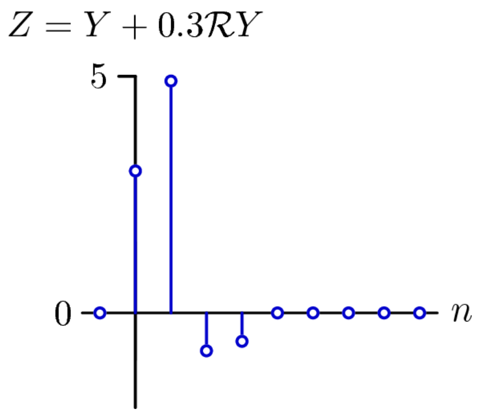
$$\text{if } Y = X_1 + X_2 \text{ then } y[n] = x_1[n] + x_2[n] \quad .$$

That is, the value of the composite signal at step n is the sum of the values of the component signals. Here are some new signals constructed by summing, scaling, and delaying the unit sample.





Note that, because each of our operations returns a signal, we can use their results again as elements in new combinations, showing that our system has true compositionality. In addition, we can abstract, by naming signals. So, for example, we might define $Y = 3\Delta + 4\mathcal{R}\Delta - 2\mathcal{R}^2\Delta$, and then make a new signal $Z = Y + 0.3\mathcal{R}Y$, which would look like this:



Be sure you understand how the heights of the spikes are determined by the definition of Z.

Exercise 5.2
Draw a picture of samples -1 through 4 of $Y - \mathcal{R}Y$.

It is important to remember that, because signals are infinite objects, these combination operations are abstract mathematical operations. You could never somehow 'make' a new signal by calculating its value at every index. It is possible, however, to calculate the value at any particular index, as it is required.

Advancing

If we allow ourselves one more operation, that of 'advancing' the signal one step (just like delaying, but in the other direction, written Ł for left-shift), then *any* signal can be composed from the unit sample, using a (possibly infinite) number of these operations. We can demonstrate this claim by construction: to define a signal V with value v_n at index n, for any set of integer n, we simply set

$$V = v_0\Delta + \sum_{n=1}^{\infty} v_n \mathcal{R}^n \Delta + \sum_{n=1}^{\infty} v_{-n} Ł^n \Delta \ \ ,$$

where $\mathcal{R}^n$ and $Ł^n$ are shorthand for applying $\mathcal{R}$ and Ł, respectively, n times.

If n represents time, then physical systems are always *causal*: inputs that arrive after time n_0 cannot affect the output before time n_0. Such systems cannot advance signals: they can be written without Ł.

5.2.3 Algebraic Properties of Operations on Signals

Adding and scaling satisfy the familiar algebraic properties of addition and multiplication: addition is commutative and associative, scaling is commutative (in the sense that it doesn't matter whether we pre- or post-multiply) and scaling distributes over addition:

$$c \cdot (X_1 + X_2) = c \cdot X_1 + c \cdot X_2 \ \ ,$$

which can be verified by defining $Y = c \cdot (X_1 + X_2)$ and $Z = c \cdot X_1 + c \cdot X_2$ and checking that $y[n] = z[n]$ for all n:

$$\begin{aligned} y[n] &= z[n] \\ c \cdot (x_1[n] + x_2[n]) &= (c \cdot x_1[n]) + (c \cdot x_2[n]) \end{aligned}$$

which clearly holds based on algebraic properties of arithmetic on real numbers.

In addition, $\mathcal{R}$ distributes over addition and scaling, so that:

$$\begin{aligned} \mathcal{R}(X_1 + X_2) &= \mathcal{R}X_1 + \mathcal{R}X_2 \\ \mathcal{R}(c \cdot X) &= c \cdot \mathcal{R}X \ \ . \end{aligned}$$

Exercise 5.3
Verify that $\mathcal{R}$ distributes over addition and multiplication by checking that the appropriate relations hold at some arbitrary step n.

These algebraic relationships mean that we can take any finite expression involving Δ, $\mathcal{R}$, $+$ and $\cdot$ and convert it into the form

$$(a_o + a_1\mathcal{R}^1 + a_2\mathcal{R}^2 + \ldots + a_N\mathcal{R}^N)\Delta \quad .$$

That is, we can express the entire signal as a *polynomial in* $\mathcal{R}$, applied to the unit sample.

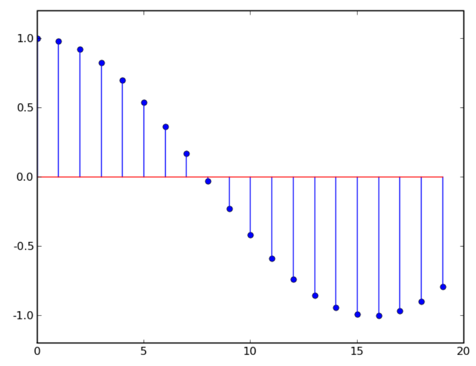
In our previous example, it means that we can rewrite $3\Delta + 4\mathcal{R}\Delta - 2\mathcal{R}^2\Delta$ as $(3 + 4\mathcal{R} - 2\mathcal{R}^2)\Delta$.

5.2.4 Sinusoidal Primitives

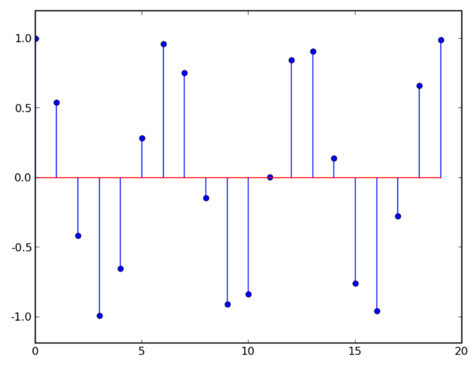
We just saw how to construct complicated signals by summing unit sample signals that are appropriately scaled and shifted. We could similarly start with a family of discretely-sampled sinusoids as our primitives, where

$$x[n] = \cos(\Omega n) \quad .$$

Here are plots of two primitives in this family:



$\cos(0.2n)$



$\cos(1.0n)$

The second plot may seem confusing, but it is just a sparsely sampled sinusoid. Note that signals constructed from even a single sinusoid have non-zero values defined at an infinity of steps; this is in contrast to signals constructed from a finite sum of scaled and shifted unit samples.

Exercise 5.4

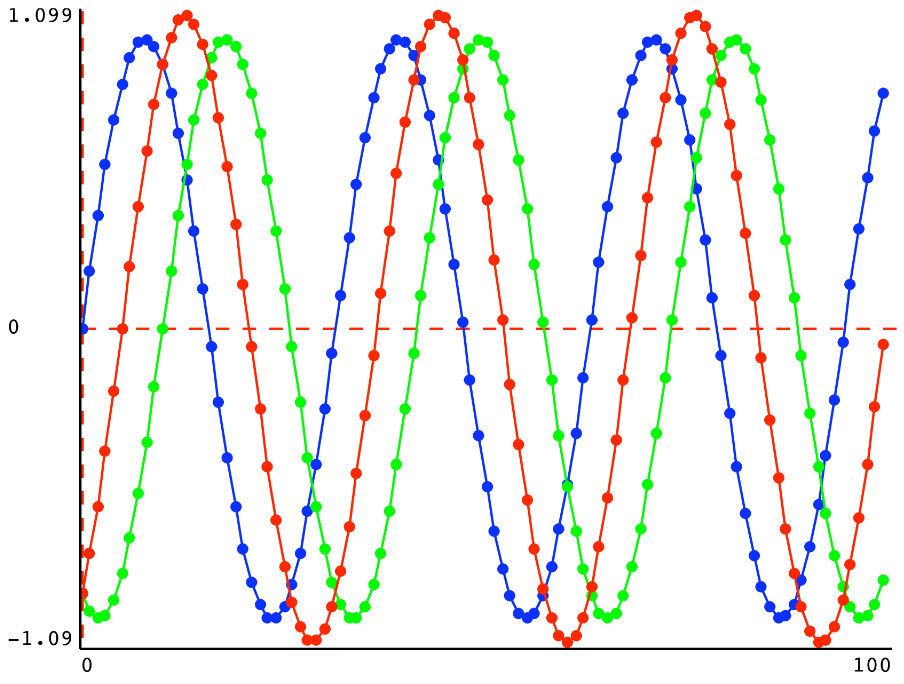
If $x[n] = cos(0.2n)$, what would be the values of $\mathcal{R}X$ at steps -3 and 5?

Here are two sinusoids and their sum, made as follows:

Exercise 5.5

$$\begin{aligned} s_1[n] &= \cos(0.2n - \pi/2) \\ S_2 &= \mathcal{R}^{10} S_1 \\ S_3 &= S_1 + S_2 \end{aligned}$$

The blue line is the S_1, the green line is the same signal, delayed by 10, which is S_2, and the red line is their sum.



5.3 Feedforward Systems

We will start by looking at a subclass of discrete-time LTI system, which are exactly those that can be described as performing some combination of scaling, delay, and addition operations on the input signal. We will develop several ways of representing such systems, and see how to combine them to get more complex systems in this same class.

5.3.1 Representing Systems

We can represent systems using operator equations, difference equations, block diagrams, and Python state machines. Each makes some things clearer and some operations easier. It is important to understand how to convert between the different representations.

Operator equation

An operator equation is a description of how signals are related to one another, using the operations of scaling, delay, and addition on whole signals.

Consider a system that has an input signal X, and whose output signal is $X - \mathcal{R}X$. We can describe that system using the operator equation

$$Y = X - \mathcal{R}X \quad .$$

Using the algebraic properties of operators on signals described in Section 5.2.3, we can rewrite this as

$$Y = (1 - \mathcal{R})X \quad ,$$

which clearly expresses a relationship between input signal X and output signal Y, whatever X may be.

Feedforward systems can always be described using an operator equation of the form

$$Y = \Phi X \quad ,$$

where Φ is a polynomial in $\mathcal{R}$.

Difference Equation

An alternative representation of the relationship between signals is a *difference equation*. A difference equation describes a relationship that holds among samples (values at particular times) of signals. We use an index n in the difference equation to refer to a particular time index, but the specification of the corresponding system is that the difference equation hold for *all* values of n.

The operator equation

$$Y = X - \mathcal{R}X \quad .$$

can be expressed as this equivalent difference equation:

$$y[n] = x[n] - x[n-1] \quad .$$

The operation of delaying a signal can be seen here as referring to a sample of that signal at time step $n - 1$.

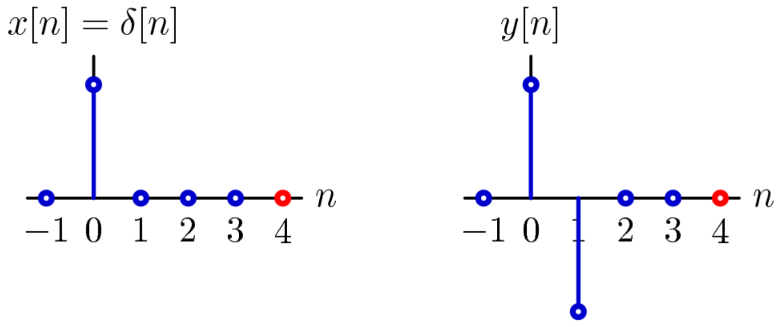
Difference equations are convenient for step-by-step analysis, letting us compute the value of an output signal at any time step, given the values of the input signal.

So, if the input signal X is the unit sample signal,

$$x[n] = \delta[n] = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases} \quad .$$

then using a difference equation, we can compute individual values of the output signal Y:

$$\begin{array}{rcccccc} y[n] & = & x[n]-x[n-1] & & & & \\ y[0] & = & x[0]-x[-1] & = & 1-0 & = & 1 \\ y[1] & = & x[1]-x[0] & = & 0-1 & = & -1 \\ y[2] & = & x[2]-x[1] & = & 0-0 & = & 0 \\ y[3] & = & x[3]-x[2] & = & 0-0 & = & 0 \\ \ldots & & & & & & \end{array}$$



Block diagrams

Another way of describing a system is by drawing a *block diagram*, which is made up of components, connected by lines with arrows on them. The lines represent signals; all lines that are connected to one another (not going through a round, triangular, or circular component) represent the same signal.

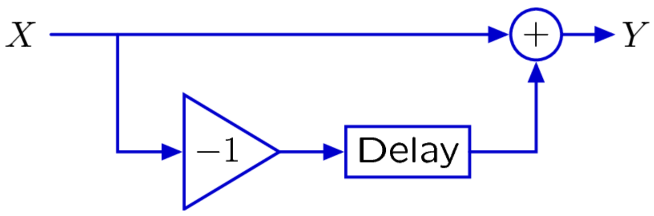
The components represent systems. There are three primitive components corresponding to our operations on signals:

- *Delay* components are drawn as rectangles, labeled `Delay`, with two lines connected to them, one with an arrow coming in and one going out. If X is the signal on the line coming into the delay, then the signal coming out is $\mathcal{R}X$.
- *Scale* (or *gain*) components are drawn as triangles, labeled with a positive or negative number c, with two lines connected to them, one with an arrow coming in and one going out. If X is the signal on the line coming into the gain component, then the signal coming out is $c \cdot X$.
- *Adder* components are drawn as circles, labeled with $+$, three lines connected to them, two with arrows coming in and one going out. If X_1 and X_2 are the signals on the lines pointing into the adder, then the signal coming out is $X_1 + X_2$.

The system

$$Y = X - \mathcal{R}X$$

can be represented with this block diagram.



State machines

Of course, since feedforward LTI systems are a type of state machine, we can make an equivalent definition using our Python state-machine specification language. So, our system

$$Y = X - \mathcal{R}X$$

can be specified in Python as a state machine by:

```
class Diff(sm.SM):
    def __init__(self, previousInput):
        self.startState = previousInput
    def getNextValues(self, state, inp):
        return (inp, inp-state)
```

Here, the `state` is the value of the previous input. One important thing to notice is that, since we have to be able to run a state machine and generate outputs, it has to start with a value for its internal state, which is the input signal's value at time -1. If we were to run:

```
Diff(0).transduce([1, 0, 0, 0])
```

we would get the result

```
[1, -1, 0, 0]
```

This same state machine can also be expressed as a combination of primitive state machines (as defined in Section 4.1.2 and Section 4.2).

```
diff = sm.ParallelAdd(sm.Wire(),
                      sm.Cascade(sm.Gain(-1), sm.R(0)))
```

Note that `sm.R` is another name for `sm.Delay` and that the desired initial output value for the system appears as the initialization argument to the `sm.R` machine.

5.3.2 Combinations of Systems

To combine LTI systems, we will use the same cascade and parallel-add operations as we had for state machines.

Cascade multiplication

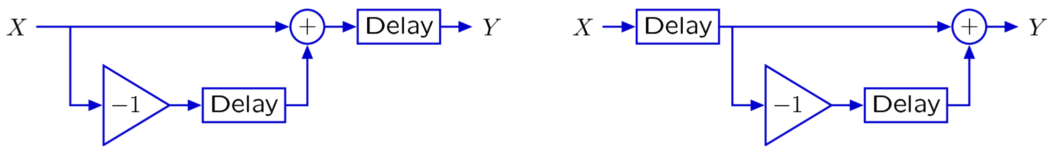
When we make a cascade combination of two systems, we let the output of one system be the input of another. So, if the system M_1 has operator equation $Y = \Phi_1 X$ and system M_2 has operator equation $Z = \Phi_2 W$, and then we compose M_1 and M_2 in cascade by setting $Y = W$, then we have a new system, with input signal X, output signal Z, and operator equation $Z = (\Phi_2 \cdot \Phi_1)X$.

The product of polynomials is another polynomial, so $\Phi_2 \cdot \Phi_1$ is a polynomial in $\mathcal{R}$. Furthermore, because polynomial multiplication is commutative, cascade combination is commutative as well (as long as the systems are *at rest*, which means that their initial states are 0).

So, for example,

$$\mathcal{R}(1 - \mathcal{R})\, X = (1 - \mathcal{R})\mathcal{R}X$$

and these two corresponding block diagrams are equivalent (the algebraic equivalence justifies the diagram equivalence):



Cascade combination, because it results in multiplication, is also associative, which means that any grouping of cascade operations on systems has the same result.

Exercise 5.6
Remembering that the condition on commutativity of cascading is that the systems start at rest, explain why machines `m3` and `m4` do not generate the same output sequence in response to the unit sample signal as input.

```
m1 = sm.ParallelAdd(sm.Wire(), sm.Cascade(sm.Gain(-1), sm.R(2)))
m2 = sm.R(3)
m3 = sm.Cascade(m1, m2)
m4 = sm.Cascade(m2, m1)
```

Parallel addition

When we make a parallel addition combination of two systems, the output signal is the sum of the output signals that would have resulted from the individual systems. So, if the system M_1 has system function $Y = \Phi_1 X$ and system M_2 has system function $Z = \Phi_2 X$, and then we compose M_1 and M_2 with parallel addition by setting output $W = Y + Z$, then we have a new system, with input signal X, output signal W, and operator equation $W = (\Phi_1 + \Phi_2)X$.

Because addition of polynomials is associative and commutative, then so is parallel addition of feed-forward linear systems.

Combining cascade and parallel operations

Finally, the distributive law applies for cascade and parallel combination, *for systems at rest*, in the same way that it applies for multiplication and addition of polynomials, so that if we have three systems, with operator equations:

$$\begin{aligned} Y &= \Phi_1 X \\ U &= \Phi_2 V \\ W &= \Phi_3 Z \ , \end{aligned}$$

and we form a cascade combination of the sum of the first two, with the third, then we have a system describable as:

$$B = (\Phi_3 \cdot (\Phi_1 + \Phi_2))A \ .$$

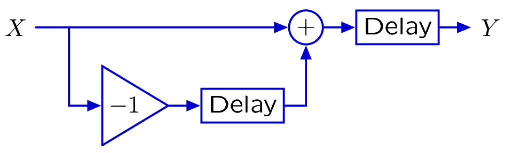
We can rewrite this, using the distributive law, as:

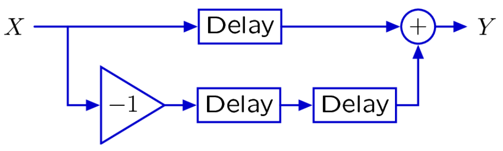
$$B = ((\Phi_3 \cdot \Phi_1) + (\Phi_3 \cdot \Phi_2))A \ .$$

So, for example,

$$\mathcal{R}(1 - \mathcal{R}) = \mathcal{R} - \mathcal{R}^2 \ ,$$

and these two corresponding block diagrams are equivalent:





<u>Exercise 5.7</u>
The first machine in the diagram above can be described, for certain initial output values as:

```
m1 = sm.Cascade(sm.ParallelAdd(sm.Wire(),
                               sm.Cascade(sm.Gain(-1), sm.Delay(2))),
                sm.Delay(3))
```

The second machine can be described as:

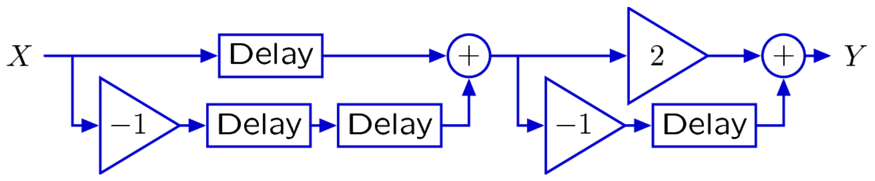
```
m2 = sm.ParallelAdd(sm.Delay(d1),
                    sm.Cascade(sm.Gain(-1),
                               sm.Cascade(sm.Delay(d2), sm.Delay(d3))))
```

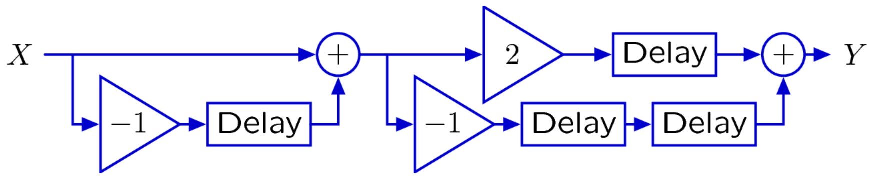
Provide values of `d1`, `d2`, and `d3` that will cause `m2` to generate the same output sequence as `m1` in response to the unit sample signal as input.

Here is another example of two equivalent operator equations

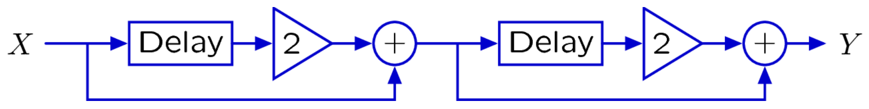
$$(\mathcal{R} - \mathcal{R}^2)(2 - \mathcal{R})X = (1 - \mathcal{R})(2\mathcal{R} - \mathcal{R}^2)X$$

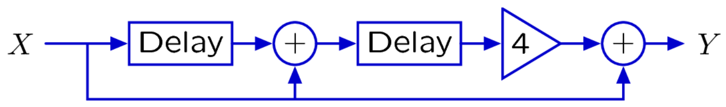
and these two corresponding block diagrams are equivalent if the systems start at rest:

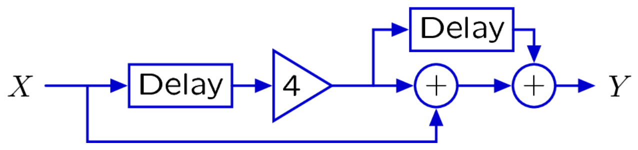




Exercise 5.8
Convince yourself that all of these systems are equivalent. One strategy is to convert them all to operator equation representation.





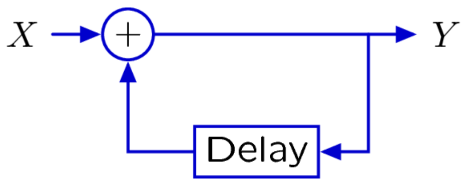


5.4 Feedback Systems

So far, all of our example systems have been *feedforward*: the dependencies have all flowed from the input through to the output, with no dependence of an output on previous output values. In this section, we will extend our representations and analysis to handle the general class of LTI systems in which the output can depend on any finite number of previous input or output values.

5.4.1 Accumulator Example

Consider this block diagram, of an *accumulator*:



It's reasonably straightforward to look at this block diagram and see that the associated difference equation is

$$y[n] = x[n] + y[n-1] \ ,$$

because the output on any given step is the sum of the input on that step and the output from the previous step.

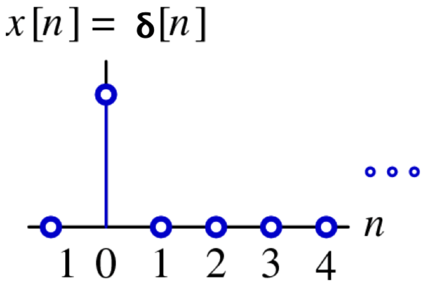
Let's use the difference equation to understand what the output of this system is when the input is the unit sample signal. To compute the output at step n, we need to evaluate

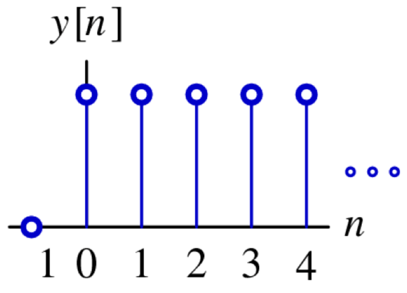
$$y[n] = x[n] + y[n-1] \ .$$

We immediately run up against a question: what is the value of $y[n-1]$? The answer clearly has a profound effect on the output of the system. In our treatment of feedback systems, we will generally assume that they start 'at rest', which means that all values of the inputs and outputs at steps less than 0 are 0. That assumption lets us fill in the following table:

$$\begin{array}{lcccccc}
y[n] & = & x[n] + y[n-1] & & & & \\
y[0] & = & x[0] + y[-1] & = & 1+0 & = & 1 \\
y[1] & = & x[1] + y[0] & = & 0+1 & = & 1 \\
y[2] & = & x[2] + y[1] & = & 0+1 & = & 1 \\
\dots & & & & & &
\end{array}$$

Here are plots of the input signal X and the output signal Y:





This result may be somewhat surprising! In feedforward systems, we saw that the output was always a finite sum of scaled and delayed versions of the input signal; so that if the input signal was transient (had a finite number of non-zero samples) then the output signal would be transient as well. But, in this feedback system, we have a transient input with a persistent (infinitely many non-zero samples) output.

We can also look at the operator equation for this system. Again, reading it off of the block diagram, it seems like it should be

$$Y = X + \mathcal{R}Y \ .$$

It's a well-formed equation, but it isn't immediately clear how to use it to determine Y. Using what we already know about operator algebra, we can rewrite it as:

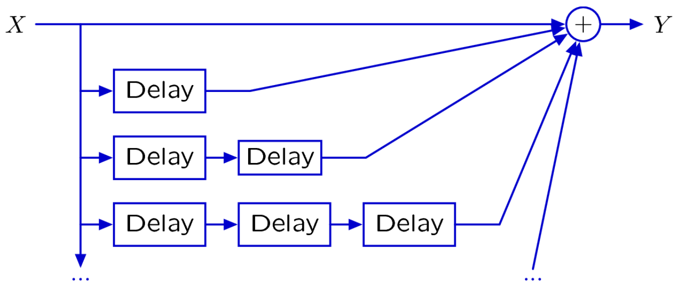
$$Y(1-\mathcal{R}) = X \ ,$$

which defines Y to be the signal such that the difference between Y and $\mathcal{R}Y$ is X. But how can we find that Y?

We will now show that we can think of the accumulator system as being equivalent to another system, in which the output is the sum of infinitely many feedforward paths, each of which delays the input by a different, fixed value. This system has an operator equation of the form

$$Y = (1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots)\, X$$

and can be represented with a block diagram of the form:



These systems are equivalent in the sense that if each is initially at rest, they will produce identical outputs from the same input. We can see this by taking the original definition and repeatedly substituting in the definition of Y in for its occurrence on the right hand side:

$$\begin{aligned} Y &= X + \mathcal{R}Y \\ Y &= X + \mathcal{R}(X + \mathcal{R}Y) \\ Y &= X + \mathcal{R}(X + \mathcal{R}(X + \mathcal{R}Y)) \\ Y &= X + \mathcal{R}(X + \mathcal{R}(X + \mathcal{R}(X + \mathcal{R}Y))) \\ Y &= (1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \ldots)X \end{aligned}$$

Now, we can informally derive a 'definition' of the reciprocal of $1 - \mathcal{R}$

$$\frac{1}{1-\mathcal{R}} = 1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots \quad .$$

In the following it will help to remind ourselves of the derivation of the formula for the sum of an infinte geometric series:

$$\begin{aligned} S &= 1 + \quad x + x^2 + x^3 + \cdots \\ Sx &= \qquad\; x + x^2 + x^3 + x^4 \cdots \end{aligned}$$

Subtracting the second equation from the first we get

$$S(1 - x) \;=\; 1$$

And so, provided $|x| < 1$,

$$\tfrac{1}{1-x} \;=\; 1 + x + x^2 + x^3 + \cdots$$

Similarly, we can consider the system O, where

$$\begin{aligned} O &= 1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots \\ O\mathcal{R} &= \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots \end{aligned}$$

So

$$O(1 - \mathcal{R}) = 1$$

And so,

$$\tfrac{1}{1-\mathcal{R}} = 1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots$$

Exercise 5.9
Check this derivation by showing that

$$(1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots)(1 - \mathcal{R}) = 1$$

So, we can rewrite the operator equation for the accumulator as

$$Y = \frac{1}{1-\mathcal{R}} X \quad .$$

We don't have an intuitive way to interpret this relationship between X and Y yet, but we will spend most of the rest of this chapter on developing intuition and analysis for systems with feedback.

5.4.2 General Form of LTI Systems

We can now treat the general case of LTI systems, including feedback. In general, LTI systems can be described by difference equations of the form:

$$\begin{aligned} y[n] &= c_0\, y[n-1] + c_1\, y[n-2] + \ldots + c_{k-1}\, y[n-k] \\ &\quad + d_0\, x[n] + d_1\, x[n-1] + \ldots + d_j\, x[n-j] \quad . \end{aligned}$$

The state of this system consists of the k previous output values and j previous input values. The output $y[n]$ is a linear combination of the k previous output values, $y[n-1], \ldots, y[n-k]$, j previous input values, $x[n-1], \ldots, x[n-j]$, and the current input, $x[n]$.

This class of state machines can be represented, in generality, in Python, using the `LTISM` class. The state is a tuple, containing a list of the j previous input values and a list of the k previous output values.

```
class LTISM (sm.SM):
    def __init__(self, dCoeffs, cCoeffs):
        j = len(dCoeffs) - 1
        k = len(cCoeffs)

        self.cCoeffs = cCoeffs
        self.dCoeffs = dCoeffs
        self.startState = ([0.0]*j, [0.0]*k)

    def getNextValues(self, state, input):
        (inputs, outputs) = state
        inputs = [input] + inputs

        currentOutput = util.dotProd(outputs, self.cCoeffs) + \
                        util.dotProd(inputs, self.dCoeffs)

        return ((inputs[:-1], ([currentOutput] + outputs)[:-1]),
                currentOutput)
```

The `util.dotProd` method takes two equal-length lists of numbers and returns the sum of their elementwise products (the dot-product of the two vectors). To keep this code easy to read, we do not handle correctly the case where `cCoeffs` is empty, though it is handled properly in our library implementation.

5.4.3 System Functions

Now, we are going to engage in a shift of perspective. We started by defining a new signal Y in terms of an old signal X, much as we might, in algebra, define $y = x + 6$. Sometimes, however, we want to speak of the relationship between x and y in the general case, without a specific x or y in mind. We do this by defining a function f: $f(x) = x + 6$. We can do the same thing with LTI systems, by defining *system functions*.

If we take the general form of an LTI system given in the previous section and write it as an operator equation, we have

$$\begin{aligned} Y &= c_0\,\mathcal{R}Y + c_1\,\mathcal{R}^2Y + \ldots + c_{k-1}\,\mathcal{R}^kY + d_0\,X + d_1\,\mathcal{R}X + \ldots + d_j\,\mathcal{R}^jX \\ &= (c_0\,\mathcal{R} + c_1\,\mathcal{R}^2 + \ldots + c_{k-1}\,\mathcal{R}^k)\,Y + (d_0 + d_1\,\mathcal{R} + \ldots + d_j\,\mathcal{R}^j)\,X\ . \end{aligned}$$

We can rewrite this as

$$(1 - c_0\,\mathcal{R} - c_1\,\mathcal{R}^2 - \ldots - c_{k-1}\,\mathcal{R}^k)\,Y = (d_0 + d_1\,\mathcal{R} + \ldots + d_j\,\mathcal{R}^j)\,X\ ,$$

So

$$\frac{Y}{X} = \frac{d_0 + d_1\mathcal{R} + d_2\mathcal{R}^2 + d_3\mathcal{R}^3 + \cdots}{1 - c_0\mathcal{R} - c_1\mathcal{R}^2 - c_2\mathcal{R}^3 - \cdots}\ ,$$

which has the form

$$\frac{Y}{X} = \frac{N(\mathcal{R})}{D(\mathcal{R})} \ ,$$

where $N(\mathcal{R})$, the numerator, is a polynomial in $\mathcal{R}$, and $D(\mathcal{R})$, the denominator, is also a polynomial in $\mathcal{R}$. We will refer to Y/X as the *system function*: it characterizes the operation of a system, independent of the particular input and output signals involved.

The system function is most typically written in the form

$$\frac{Y}{X} = \frac{b_0 + b_1\mathcal{R} + b_2\mathcal{R}^2 + b_3\mathcal{R}^3 + \cdots}{a_0 + a_1\mathcal{R} + a_2\mathcal{R}^2 + a_3\mathcal{R}^3 + \cdots} \ ,$$

where $c_i = -a_{i+1}/a_0$ and $d_i = b_i/a_0$. It can be completely characterized by the coefficients of the denominator polynomial, a_i, and the coefficients of the numerator polynomial, b_i. It is always possible to rewrite this in a form in which $a_0 = 1$.

Feedforward systems have no dependence on previous values of Y, so they have $D(\mathcal{R}) = 1$. Feedback systems have persistent behavior, which is determined by $D(\mathcal{R})$. We will study this dependence in detail in Section 5.5.

5.4.4 Primitive Systems

Just as we had a PCAP system for signals, we have one for LTI system, in terms of system functions, as well. We can specify system functions for each of our system primitives.

A *gain* element is governed by operator equation $Y = kX$, for constant k, so its system function is

$$H = \frac{Y}{X} = k \ .$$

A *delay* element is governed by operator equation $Y = \mathcal{R}X$, so its system function is

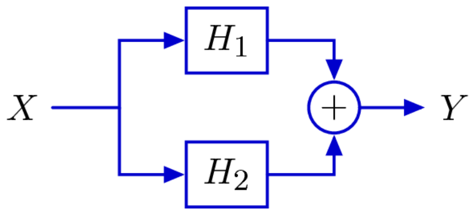
$$H = \frac{Y}{X} = \mathcal{R} \ .$$

5.4.5 Combining System Functions

We have three basic composition operations: sum, cascade, and feedback. This PCAP system, as our previous ones have been, is *compositional*, in the sense that whenever we make a new system function out of existing ones, it is a system function in its own right, which can be an element in further compositions.

5.4.5.1 Addition

The system function of the *sum* of two systems is the sum of their system functions. So, given two systems with system functions H_1 and H_2, connected like this:



and letting

$$Y_1 = H_1 X \quad \text{and} \quad Y_2 = H_2 X \ ,$$

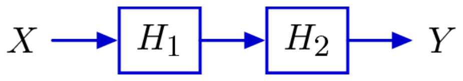
we have

$$\begin{aligned} Y &= Y_1 + Y_2 \\ &= H_1 X + H_2 X \\ &= (H_1 + H_2) X \\ &= H X \ , \end{aligned}$$

where $H = H_1 + H_2$.

5.4.5.2 Cascade

The system function of the *cascade* of two systems is the product of their system functions. So, given two systems with system functions H_1 and H_2, connected like this:



and letting

$$W = H_1 X \quad \text{and} \quad Y = H_2 W \ ,$$

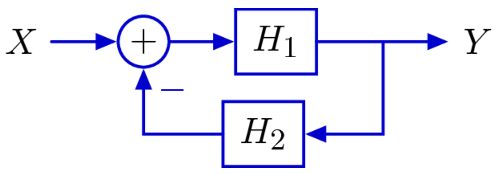
we have

$$\begin{aligned} Y &= H_2 W \\ &= H_2 H_1 X \\ &= H X \ , \end{aligned}$$

where $H = H_2 H_1$. And note that, as was the case with purely feedforward systems, cascade combination is still commutative, so it doesn't matter whether H_1 or H_2 comes first in the cascade. This surprising fact holds because we are only considering LTI systems *starting at rest*; for more general classes of systems, such as the general class of state machines we have worked with before, the ordering of a cascade *does* matter.

5.4.5.3 Feedback

Here we study a particular way of connecting systems in a *negative feedback* combination. Consider two systems connected like this



and *pay careful attention to the negative sign on the feedback input to the addition.* It is really just shorthand; the negative sign could be replaced with a gain component with value -1. This negative feedback arrangement is frequently used to model a case in which X is a desired value for some signal and Y is its actual value; thus the input to H_1 is the difference between the desired actual values, often called an *error signal.* We can simply write down the operator equation governing this system and use standard algebraic operations to determine the system function:

$$\begin{aligned}
Y &= H_1(X - H_2Y) \\
Y + H_1H_2Y &= H_1X \\
Y(1 + H_1H_2) &= H_1X \\
Y &= \tfrac{H_1}{1+H_1H_2}X \\
Y &= HX \ ,
\end{aligned}$$

where

$$H = \frac{H_1}{1 + H_1H_2} \ .$$

Armed with this set of primitives and composition methods, we can specify a large class of machines.

5.5 Predicting System Behavior

We have seen how to construct complex discrete-time LTI systems; in this section we will see how we can use properties of the system function to predict how the system will behave, in the long term, and for any input. We will start by analyzing simple systems and then move to more complex ones.

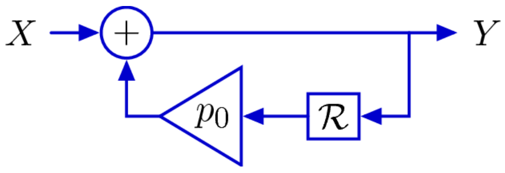
We can provide a general characterization of the long-term behavior of the output, as increasing or decreasing, with constant or alternating sign, for any finite input to the system. We will begin by studying the *unit-sample* response of systems, and then generalize to more general input signals; similarly, we will begin by studying simple systems and generalize to more complex ones.

5.5.1 First-Order Systems

Systems that only have forward connections can only have a finite response; that means that if we put in a unit sample (or other signal with only a finite number of non-zero samples) then the output signal will only have a finite number of non-zero samples.

Systems with feedback have a surprisingly different character. Finite inputs can result in *persistent response*; that is, in output signals with infinitely many non-zero samples. Furthermore, the qualitative long-term behavior of this output is generally independent of the particular input given to the system, for any finite input. In this section, we will consider the class of *first-order* systems, in which the denominator of the system function is a first-order polynomial (that is, it only involves $\mathcal{R}$, but not $\mathcal{R}^2$ or other higher powers of $\mathcal{R}$.)

Let's consider this very simple system



for which we can write an operator equation

$$\begin{aligned} Y &= X + p_0\mathcal{R}Y \\ (1 - p_0\mathcal{R})Y &= X \\ Y &= \frac{X}{1 - p_0\mathcal{R}} \end{aligned}$$

and derive a system function

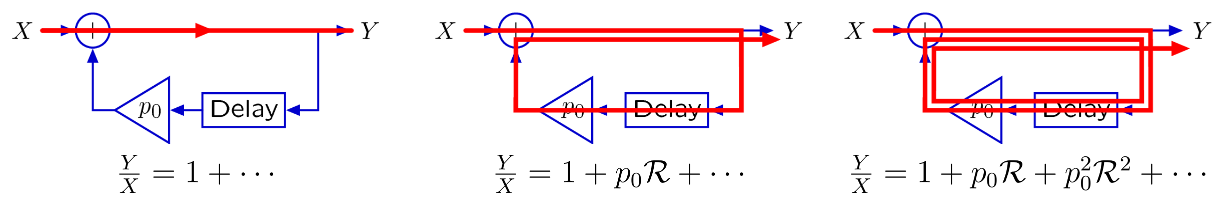
$$H = \frac{Y}{X} = \frac{1}{1 - p_0\mathcal{R}} \quad .$$

Recall the infinite series representation of this system function (derived in Section 5.4.1):

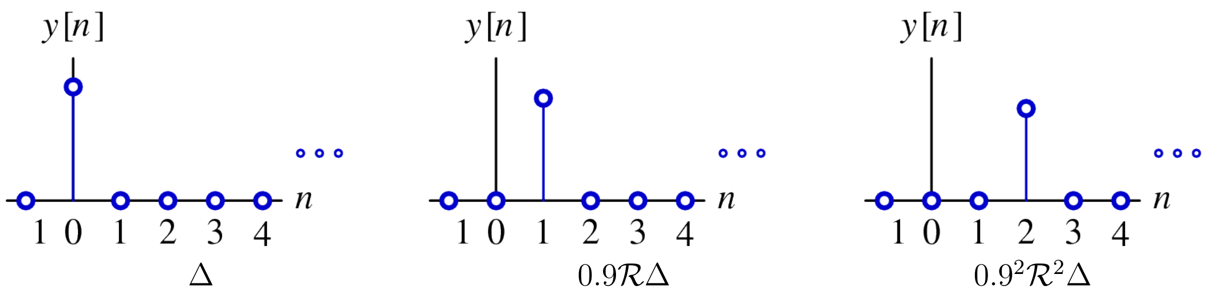
$$\frac{1}{1 - p_0\mathcal{R}} = 1 + p_0\mathcal{R} + p_0^2\mathcal{R}^2 + p_0^3\mathcal{R}^3 + p_0^4\mathcal{R}^4 + \cdots \quad .$$

We can make intuitive sense of this by considering how the signal flows through the system. On each step, the output of the system is being fed back into the input. Consider the simple case where the input is the unit sample ($X = \Delta$). Then, after step 0, when the input is 1, there is no further input, and the system continues to respond.

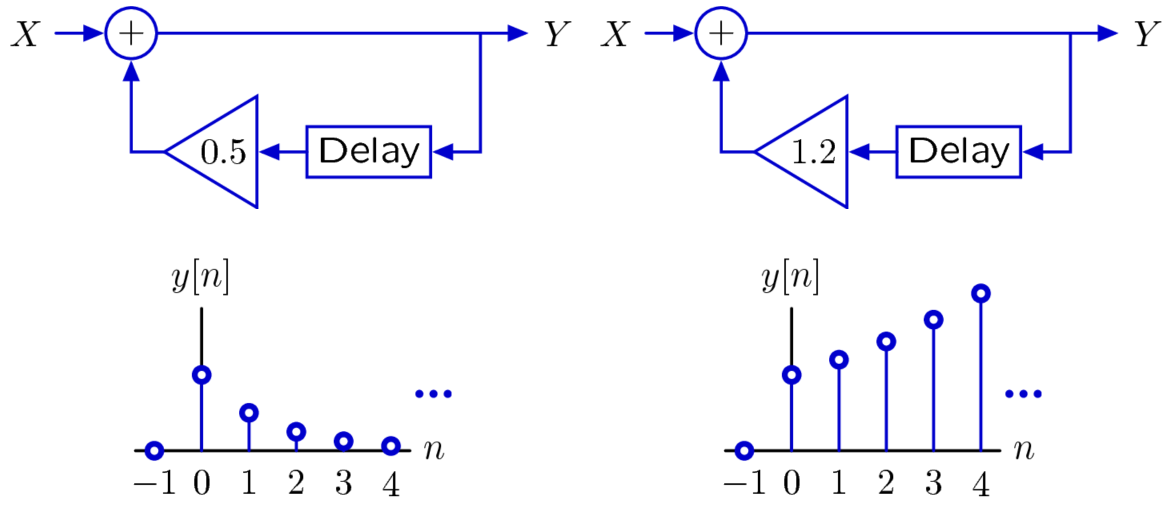
In this table, we see that the whole output signal is a sum of scaled and delayed copies of the input signal; the bottom row of figures shows the first three terms in the infinite sum of signals, for the case where $p_0 = 0.9$.



X
Y
p_0
Delay
$\frac{Y}{X} = 1 + \cdots$
$\frac{Y}{X} = 1 + p_0\mathcal{R} + \cdots$
$\frac{Y}{X} = 1 + p_0\mathcal{R} + p_0^2\mathcal{R}^2 + \cdots$



If traversing the cycle decreases or increases the magnitude of the signal, then the sample values will decay or grow, respectively, as time increases.

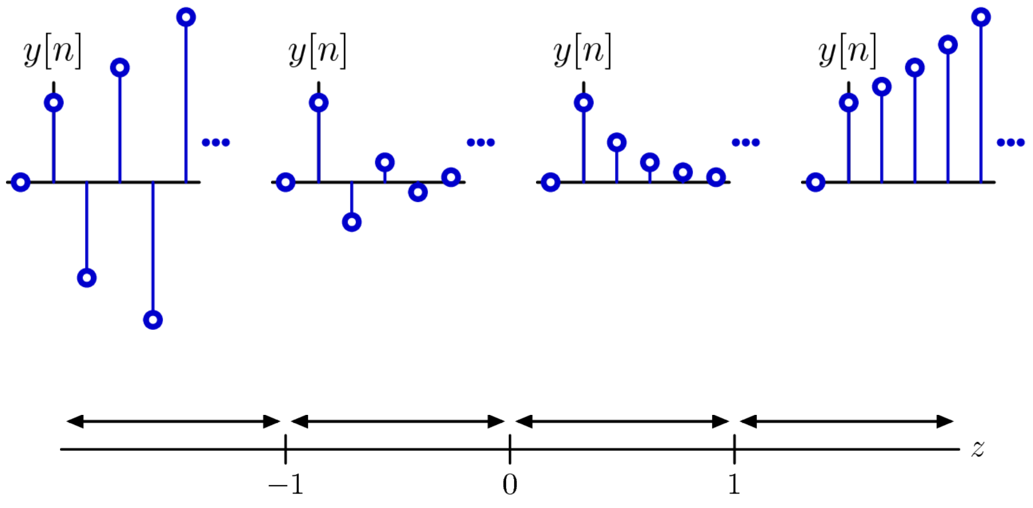


For the first system, the unit sample response is $y[n] = (0.5)^n$; for the second, it's $y[n] = (1.2)^n$.

These system responses can be characterized by a single number, called the *pole*, which is the base of the geometric sequence. The value of the pole, p_0, determines the nature and rate of growth.

- If $p_0 < -1$, the magnitude increases to infinity and the sign alternates.
- If $-1 < p_0 < 0$, the magnitude decreases and the sign alternates.
- If $0 < p_0 < 1$, the magnitude decreases monotonically.
- If $p_0 > 1$, the magnitude increases monotonically to infinity.

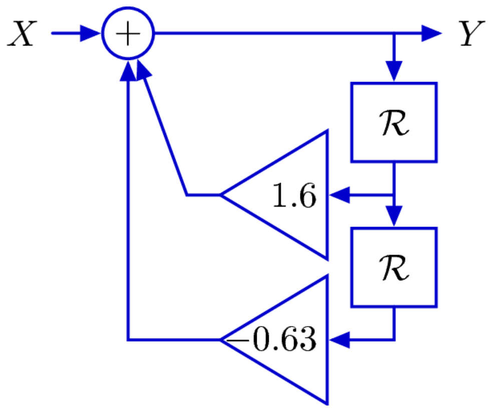
Any system for which the output magnitude increases to infinity, whether monotonically or not, is called *unstable*. Systems whose output magnitude decreases or stays constant are called *stable*.



5.5.2 Second-Order Systems

We will call these persistent long-term behaviors of a signal (and, hence, of the system that generates such signals) *modes*. For a fixed p_0, the first-order system only exhibited one mode (but different values of p_0 resulted in very different modes). As we build more complex systems, they will have multiple modes, which manifest as more complex behavior. Second-order systems are characterized by a system function whose denominator polynomial is second order; they will generally exhibit two modes. We will find that it is the mode whose pole has the largest magnitude that governs the long-term behavior of the system. We will call this pole the *dominant pole*.

Consider this system



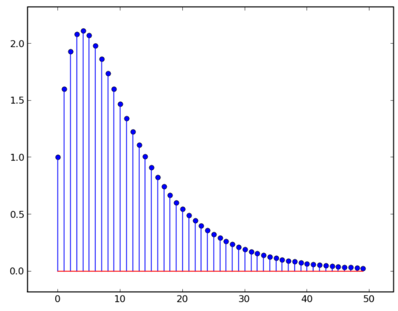
We can describe it with the operator equation

$$Y = 1.6\mathcal{R}Y - 0.63\mathcal{R}^2Y + X \quad ,$$

so the system function is

$$H = \frac{1}{1 - 1.6\mathcal{R} + 0.63\mathcal{R}^2} \quad .$$

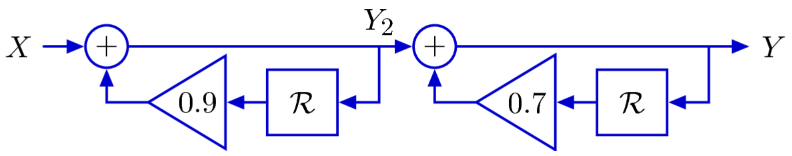
Here is its response to a unit sample signal:

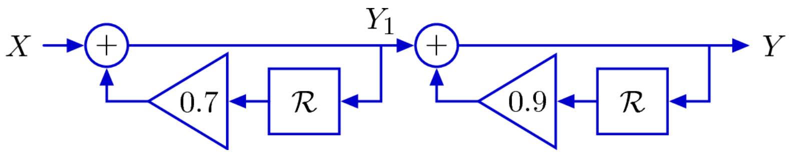


We can try to understand its behavior by decomposing it in different ways. First, let's see if we can see it as a cascade of two systems. To do so, we need to find H_1 and H_2 such that $H_1 H_2 = H$. We can do that by factoring H to get

$$H_1 = \frac{1}{1 - 0.7\mathcal{R}} \qquad \text{and} \qquad H_2 = \frac{1}{1 - 0.9\mathcal{R}} \quad .$$

So, we have two equivalent version of this system, describable as cascades of two systems, one with $p_0 = 0.9$ and one with $p_0 = 0.7$:





This decomposition is interesting, but it does not yet let us understand the behavior of the system as the combination of the behaviors of two subsystems.

5.5.2.1 Additive Decomposition

Another way to try to decompose the system is as the *sum* of two simpler systems. In this case, we seek H_1 and H_2 such that $H_1 + H_2 = H$. We can do a partial fraction decomposition (don't worry if you don't remember the process for doing this...we won't need to solve problems like this in detail). We start by factoring, as above, and then figure out how to decompose into additive terms:

$$\begin{aligned} H &= \frac{1}{1 - 1.6\mathcal{R} + 0.63\mathcal{R}^2} \\ &= \frac{1}{(1 - 0.9\mathcal{R})(1 - 0.7\mathcal{R})} \\ &= \frac{A}{1 - 0.9\mathcal{R}} + \frac{B}{1 - 0.7\mathcal{R}} \\ &= H_1 + H_2 \ . \end{aligned}$$

To find values for A and B, we start with

$$\frac{1}{(1 - 0.9\mathcal{R})(1 - 0.7\mathcal{R})} = \frac{A}{1 - 0.9\mathcal{R}} + \frac{B}{1 - 0.7\mathcal{R}} \ ,$$

multiply through by $1 - 1.6\mathcal{R} + 0.63\mathcal{R}^2$ to get

$$1 = A(1 - 0.7\mathcal{R}) + B(1 - 0.9\mathcal{R}) \ ,$$

and collect like terms:

$$1 = (A + B) - (0.7A + 0.9B)\mathcal{R} \ .$$

Equating the terms that involve equal powers of $\mathcal{R}$ (including constants as terms that involve $\mathcal{R}^0$), we have:

$$\begin{aligned} 1 &= A + B \\ 0 &= 0.7A + 0.9B \ . \end{aligned}$$

Solving, we find $A = 4.5$ and $B = -3.5$, so

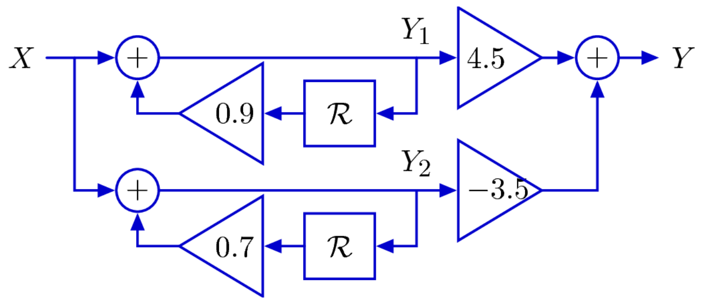
$$\frac{Y}{X} = \frac{4.5}{1 - 0.9\mathcal{R}} + \frac{-3.5}{1 - 0.7\mathcal{R}} \ ,$$

where

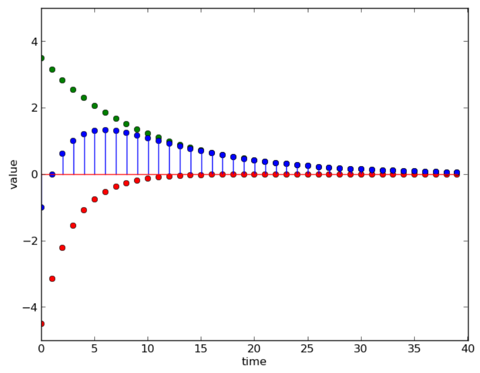
$$H_1 = \frac{4.5}{1 - 0.9\mathcal{R}} \quad \text{and} \quad H_2 = \frac{-3.5}{1 - 0.7\mathcal{R}} \ .$$

<u>Exercise 5.10</u>
Verify that $H_1 + H_2 = H$.

Here is (yet another) equivalent block diagram for this system, highlighting its decomposition into a sum:



We can understand this by looking at the responses of H_1 and of H_2 to the unit sample, and then summing them to recover the response of H to the unit sample. In the next figure, the blue stem plot is the overall signal, which is the sum of the green and red signals, which correspond to the top and bottom parts of the diagram, respectively.

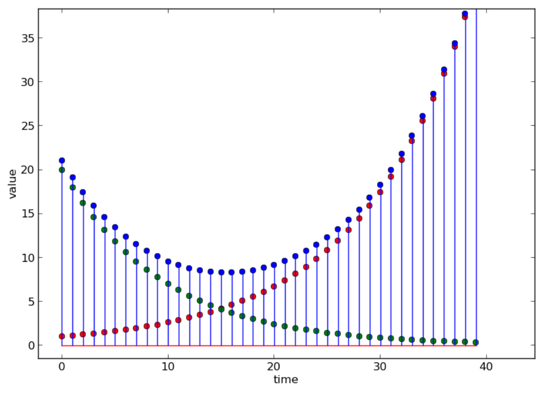


In this case, both of the poles (0.9 and 0.7) are less than 1, so the magnitude of the responses they generate decreases monotonically; their sum does not behave monotonically, but there is a time step at which the dominant pole completely dominates the other one, and the convergence is monotonic after that.

If, instead, we had a system with system function

$$H = \frac{Y}{X} = \frac{1}{1 - 1.1\mathcal{R}} + \frac{20}{1 - 0.9\mathcal{R}} \quad ,$$

what would the unit-sample response be? The first mode (first term of the sum) has pole 1.1, which means it will generate monotonically increasing output values. The second mode has pole 0.9, and will decrease monotonically. The plot below illustrates the sum of these two components:



The red points correspond to the output from the mode with pole 1.1; the green points correspond to the output from the mode with pole 0.9; and the blue stem plot shows the sum.

5.5.2.2 Complex Poles

Consider a system described by the operator equation:

$$Y = 2X + 2\mathcal{R}X - 2\mathcal{R}Y - 4\mathcal{R}^2Y \ .$$

It has system function

$$\frac{Y}{X} = \frac{2 + 2\mathcal{R}}{1 + 2\mathcal{R} + 4\mathcal{R}^2} \ . \tag{5.1}$$

But now, if we attempt to perform an additive decomposition on it, we find that the denominator cannot be factored to find real poles. Instead, we find that

$$\begin{aligned}\frac{Y}{X} &= \frac{2 + 2\mathcal{R}}{(1 - (-1 + \sqrt{-3})\mathcal{R})(1 - (-1 - \sqrt{-3})\mathcal{R})} \\ &\approx \frac{2 + 2\mathcal{R}}{(1 - (-1 + 1.732j)\mathcal{R})(1 - (-1 - 1.732j)\mathcal{R})} \ .\end{aligned} \tag{5.2}$$

So, the poles are $-1 + 1.732j$ and $-1 - 1.732j$. Note that we are using j to signal the imaginary part of a complex number.[3] What does that mean about the behavior of our system? Are the outputs real or complex? Do the output values grow or shrink with time?

Difference equations that represent physical systems have real-valued coefficients. For instance, a bank account with interest ρ might be described with difference equation

$$y[n] = (1 + \rho)y[n-1] + x[n] \ .$$

Difference equations with real-valued coefficients generate real-valued outputs from real-valued inputs. But, like the difference equation

$$y[n] = 2x[n] + 2x[n-1] - 2y[n-1] - 4y[n-1] \ ,$$

[3] Another mathematical convention uses i instead.

corresponding to system function 5.1, they might still have complex poles.

Polar representation of complex numbers

Sometimes it's easier to think about a complex number $a + bj$ instead as $re^{j\Omega}$, where

$$\begin{aligned} a &= r\cos\Omega \\ b &= r\sin\Omega \end{aligned}$$

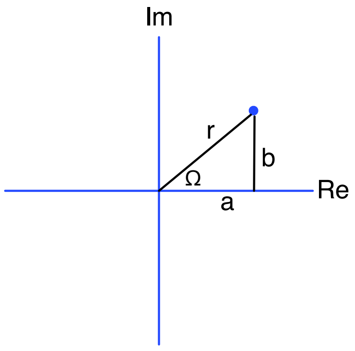
so that the *magnitude*, r, sometimes written as $|a + bj|$, is defined as

$$r = \sqrt{a^2 + b^2}$$

and the *angle*, Ω, is defined as

$$\Omega = \tan^{-1}(b, a) \quad .$$

So, if we think of (a, b) as a point in the complex plane, then (r, Ω) is its representation in polar coordinates.



This representation is justified by Euler's equation

$$e^{xj} = \cos x + j\sin x \quad ,$$

which can be directly derived from series expansions of e^z, $\sin z$ and $\cos z$. To see that this is reasonable, let's take our number, represent it as a complex exponential, and then apply Euler's equation:

$$\begin{aligned} a + bj &= re^{j\Omega} \\ &= r(\cos\Omega + j\sin\Omega) \\ &= \sqrt{a^2+b^2}(\cos(\tan^{-1}(b,a)) + j\sin(\tan^{-1}(b,a))) \\ &= \sqrt{a^2+b^2}(\frac{a}{\sqrt{a^2+b^2}} + j\frac{b}{\sqrt{a^2+b^2}}) \\ &= a + bj \end{aligned}$$

Why should we bother with this change of representation? There are some operations on complex numbers that are much more straightforwardly done in the exponential representation. In particular, let's consider raising a complex number to a power. In the

Cartesian representation, we get complex trigonometric polynomials. In the exponential representation, we get, in the quadratic case,

$$\left(re^{j\Omega}\right)^2 = re^{j\Omega}re^{j\Omega} = r^2e^{j2\Omega}.$$

More generally, we have that

$$\left(re^{j\Omega}\right)^n = r^n e^{jn\Omega} \quad ,$$

which is much tidier. This is an instance of an important trick in math and engineering: changing representations. We will often find that representing something in a different way will allow us to do some kinds of manipulations more easily. This is why we use diagrams, difference equations, operator equations and system functions all to describe LTI systems. There is no *one* best representation; each has advantages under some circumstances (and disadvantages under others).

Complex modes

Now, we're equipped to understand how complex poles of a system generate behavior: they produce complex-valued modes. Remember that we can characterize the behavior of a mode as

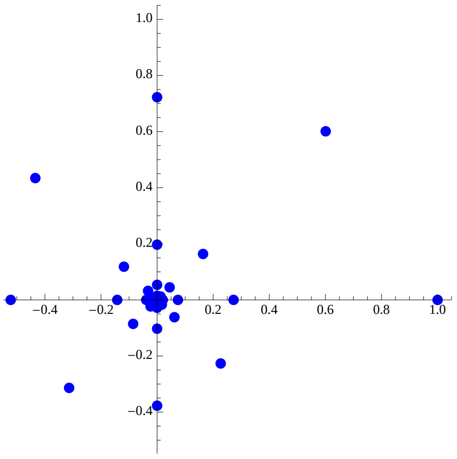
$$\frac{1}{1 - p\mathcal{R}} = 1 + p\mathcal{R} + p^2\mathcal{R}^2 + \cdots + p^n\mathcal{R}^n + \cdots \quad .$$

For a complex pole $p = re^{j\Omega}$, $p^n = r^n e^{j\Omega n}$. So

$$\frac{1}{1 - re^{j\Omega}\mathcal{R}} = 1 + re^{j\Omega}\mathcal{R} + r^2e^{j2\Omega}\mathcal{R}^2 + \cdots$$

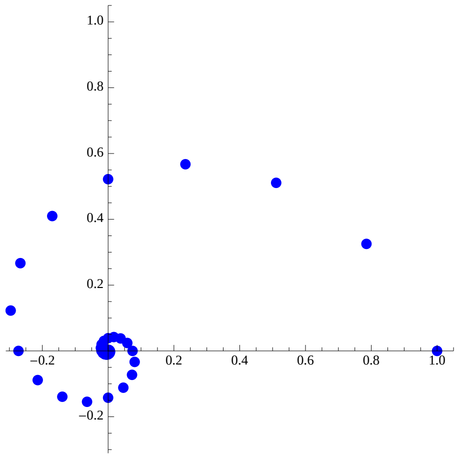
What happens as n tends to infinity when p is complex? Think of p^n as a point in the complex plane with coordinates $(r^n, \Omega n)$. The radius, r^n, will grow or shrink depending on the mode's magnitude, r. And the angle, Ωn, will simply rotate, but will not affect the magnitude of the resulting value. Note that each new point in the sequence p^n will be rotated by Ω from the previous one. We will say that the *period* of the output signal is $2\pi/\Omega$; that is the number of samples required to move through 2π radians (although this need not actually be an integer).

The sequence spirals around the origin of the complex plane. Here is a case for $r = 0.85, \Omega = \pi/4$, that spirals inward, rotating $\pi/4$ radians on each step:

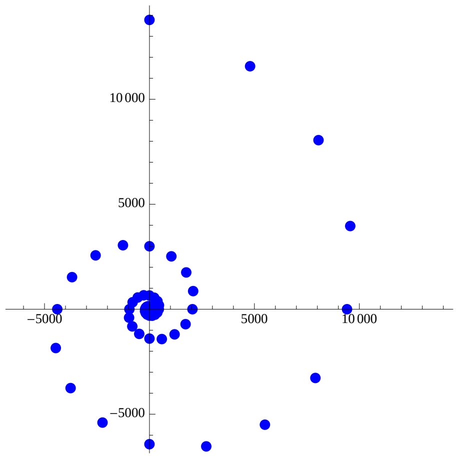


$$p = 0.85e^{j\pi/4} \approx 0.601 + 0.601j$$

Here, for example, are plots of two other complex modes, with different magnitudes and rates of oscillation. In all cases, we are plotting the unit-sample response, so the first element in the series is the real value 1. The first signal clearly converges in magnitude to 0; the second one diverges (look carefully at the scales).



$p = 0.85e^{j\pi/8} \approx 0.785 + 0.325j$



$p = 1.1e^{j\pi/8} \approx 1.016 + 0.421j$

If complex modes occurred all by themselves, then our signals would have complex numbers in them. But isolated complex pole can result only from a difference equation with complex-valued coefficients. For example, to end up with this system function

$$\frac{Y}{X} = \frac{1}{1 - re^{j\Omega}\mathcal{R}} \ ,$$

we would need to have this difference equation, which has a complex coefficient.

$$y[n] - re^{j\Omega}y[n-1] = x[n]$$

But, we have difference equations with real parameters, so we can express the value of every sample of the output signal as a linear combination of inputs and previous values with real coefficients, so we know the output signal is real-valued at all samples. The reason this all works out is that, for polynomials with real coefficients, the complex poles always come in *conjugate pairs*: that is, pairs of complex numbers $p = a + bj$ and $p^* = a - bj$ (see Section 5.5.4). In the polar representation, the conjugate pair becomes

$$\begin{aligned} a \pm bj &= \sqrt{a^2+b^2}\; e^{j\tan^{-1}(\pm b,a)} \\ &= \sqrt{a^2+b^2}\; e^{\pm j\tan^{-1}(b,a)} \end{aligned}$$

where the only difference is in the sign of the angular part.

If we look at a second-order system with complex-conjugate poles, the resulting polynomial has real-valued coefficients. To see this, consider a system with poles $re^{j\Omega}$ and $re^{-j\Omega}$, so that

$$\frac{Y}{X} = \frac{1}{(1 - re^{j\Omega}\mathcal{R})(1 - re^{-j\Omega}\mathcal{R})}$$

Let's slowly multiply out the denominator:

$$\begin{aligned} (1 - re^{j\Omega}\mathcal{R})(1 - re^{-j\Omega}\mathcal{R}) &= 1 - re^{j\Omega}\mathcal{R} - re^{-j\Omega}\mathcal{R} + r^2 e^{j\Omega}e^{-j\Omega}\mathcal{R}^2 \\ &= 1 - r(e^{j\Omega} + e^{-j\Omega})\mathcal{R} + r^2 e^{j\Omega - j\Omega}\mathcal{R}^2 \\ &= 1 - r(\cos\Omega + j\sin\Omega + \cos(-\Omega) + j\sin(-\Omega))\mathcal{R} + r^2\mathcal{R}^2 \\ &= 1 - r(\cos\Omega + j\sin\Omega + \cos\Omega - j\sin\Omega)\mathcal{R} + r^2\mathcal{R}^2 \\ &= 1 - 2r\cos\Omega\mathcal{R} + r^2\mathcal{R}^2 \end{aligned}$$

(In the third line above, we use the definition of the complex exponential $e^{jx} = \cos x + j\sin x$, and on the fourth line, we use the trigonometric identities $\sin -x = -\sin x$ and $\cos -x = \cos x$.) ' So,

$$\frac{Y}{X} = \frac{1}{1 - 2r\cos\Omega\mathcal{R} + r^2\mathcal{R}^2} \; .$$

This is pretty cool! All of the imaginary parts cancel, and we are left with a system function with only real coefficients, which corresponds to the difference equation

$$y[n] = x[n] + 2r\cos\Omega\; y[n-1] - r^2 y[n-2] \; .$$

Additive decomposition with complex poles

To really understand these complex modes and how they combine to generate the output signal, we need to do an additive decomposition. That means doing a partial fraction expansion of

$$\frac{Y}{X} = \frac{1}{(1 - re^{j\Omega}\mathcal{R})(1 - re^{-j\Omega}\mathcal{R})} \; .$$

It's a little trickier than before, because we have complex numbers, but the method is the same. The end result is that we can decompose this system additively to get

$$\frac{Y}{X} = \frac{\frac{1}{2}(1 - j\cot\Omega)}{1 - re^{j\Omega}\mathcal{R}} + \frac{\frac{1}{2}(1 + j\cot\Omega)}{1 - re^{-j\Omega}\mathcal{R}} \; .$$

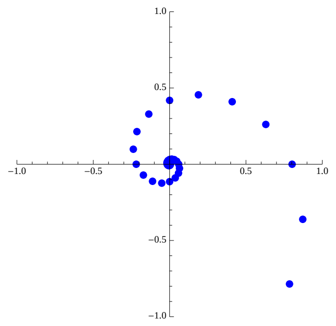
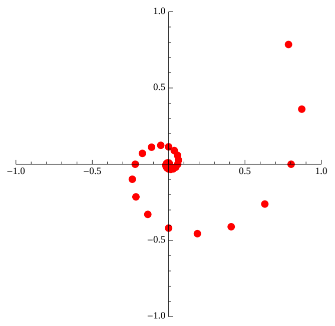
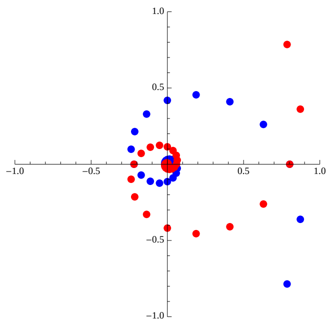
What is the unit-sample response of each of these modes? What is the unit-sample response of their sum? This might be making you nervous...it's hard to see how everything is going to come out to be real in the end.

But, let's examine the response of the additive decomposition; it's the sum of the outputs of the component systems. So, if $x[n] = \delta[n]$,

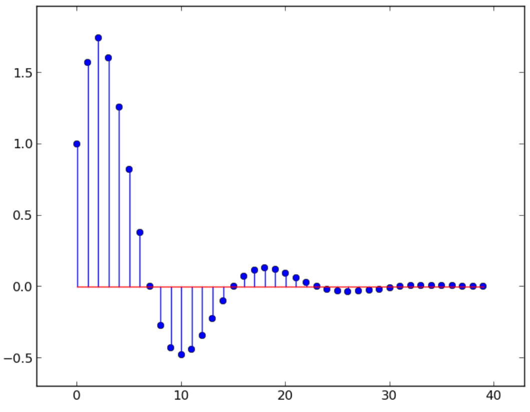
$$\begin{aligned} y[n] &= \frac{1}{2}(1 - j\cot\Omega)r^n e^{jn\Omega} + \frac{1}{2}(1 + j\cot\Omega)r^n e^{-jn\Omega} \\ &= r^n(\cos n\Omega + \cot\Omega \sin n\Omega) \ , \end{aligned} \tag{5.3}$$

which is entirely real.

The figures below show the modes for a system with poles $0.85e^{j\pi/8}$ and $0.85e^{-j\pi/8}$: the blue series starts at $\frac{1}{2}(1 - j\cot(\pi/8))$ and traces out the unit sample response of the first component; the second red series starts at $\frac{1}{2}(1 + j\cot(\pi/8))$ and traces out the unit sample response of the second component.

Note, in the third figure, that the imaginary parts of the contributions of each of the modes cancel out, and that real parts are equal. Thus, the real part of the output is going to be twice the real part of these elements. The figure below shows the unit sample response of the entire system.



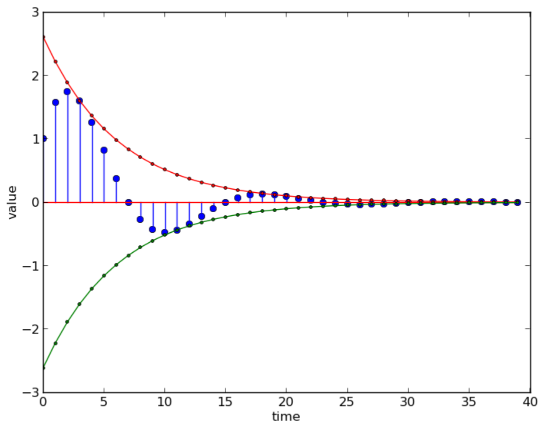
Importance of magnitude and period

Both r and Ω tells us something very useful about the way in which the system behaves. In the previous section, we derived an expression for the samples of the unit sample response for a system with a pair of complex poles. It has the form

$$y[n] = r^n(\cos n\Omega + \alpha \sin n\Omega) \ ,$$

where α is a constant.

The value of r governs the rate of exponential decrease. The value of Ω governs the rate of oscillation of the curve. It will take $2\pi/\Omega$ (the *period* of the oscillation) samples to go from peak to peak of the oscillation.[4] In our example, $\Omega = \pi/8$, so the period is 16; you should be able to count 16 samples from peak to peak.



5.5.2.3 Poles and Behavior: Summary

In a second-order system, if we let p_0 be the pole with *the largest magnitude*, then there is a time step at which the behavior of the dominant pole begins to dominate; *after that time step*

- If p_0 is real and
  - $p_0 < -1$, the magnitude increases to infinity and the sign alternates.
  - $-1 < p_0 < 0$, the magnitude decreases and the sign alternates.
  - $0 < p_0 < 1$, the magnitude decreases monotonically.
  - $p_0 > 1$, the magnitude increases monotonically to infinity.
- If p_0 is complex
  - and $|p_0| < 1$, the magnitude decreases monotonically.

[4]We are being informal here, in two ways. First, the signal does not technically have a period, because unless $r = 1$, it doesn't return to the same point. Second, unless $r = 1$, then distance from peak to peak is not exactly $2\pi/\Omega$, however, for most signals, it will give a good basis for estimating Ω.

  - and $|p_0| > 1$, the magnitude increases monotonically to infinity.

- If p_0 is complex and Ω is its angle, then the signal will be periodic, with period $2\pi/\Omega$.

Remember that any system for which the output magnitude increases to infinity, whether monotonically or not, is called *unstable*. Systems whose output magnitude decreases or stays constant are called *stable*.

As we have seen in our examples, when we add multiple modes, it is the mode with the largest pole, that is, the *dominant pole* that governs the long-term behavior.

5.5.3 Higher-Order Systems

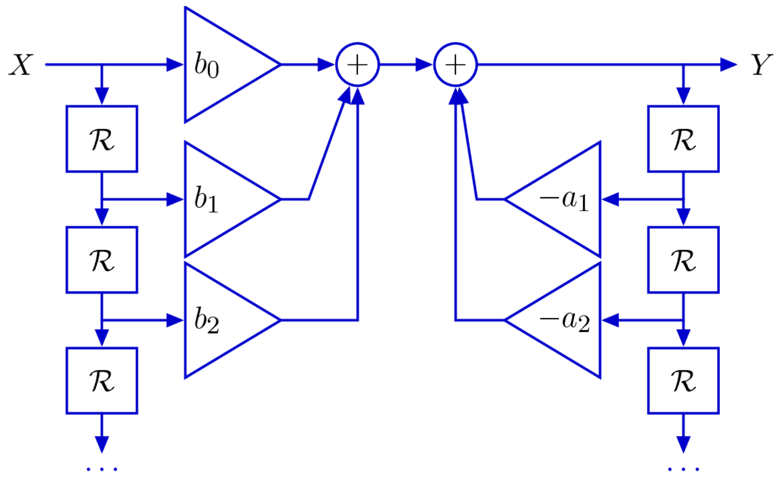
Recall that we can describe any system in terms of a system function that is the ratio of two polynomials in $\mathcal{R}$ (and assuming $a_0 = 1$):

$$\frac{Y}{X} = \frac{b_0 + b_1\mathcal{R} + b_2\mathcal{R}^2 + b_3\mathcal{R}^3 + \cdots}{1 + a_1\mathcal{R} + a_2\mathcal{R}^2 + a_3\mathcal{R}^3 + \cdots}$$

Regrouping terms, we can write this as the operator equation:

$$Y = \left(b_0 + b_1\mathcal{R} + b_2\mathcal{R}^2 + b_3\mathcal{R}^3 + \cdots\right) X - \left(a_1\mathcal{R} + a_2\mathcal{R}^2 + a_3\mathcal{R}^3 + \cdots\right) Y$$

and construct an equivalent block diagram:



Returning to the general polynomial ratio

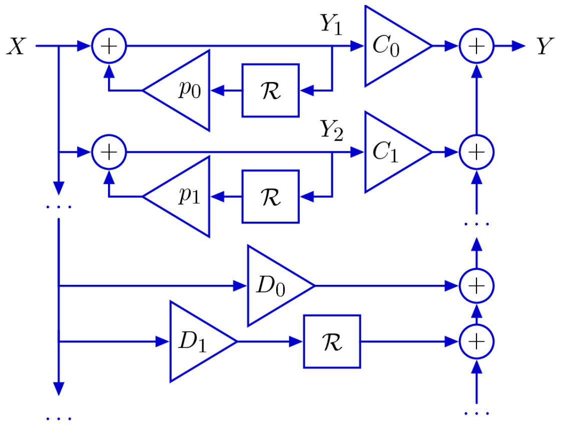
$$\frac{Y}{X} = \frac{b_0 + b_1\mathcal{R} + b_2\mathcal{R}^2 + b_3\mathcal{R}^3 + \cdots}{1 + a_1\mathcal{R} + a_2\mathcal{R}^2 + a_3\mathcal{R}^3 + \cdots}$$

We can factor the denominator of an nth-order polynomial into n factors, and then perform a partial fraction expansion, to turn it into the form of a sum of terms. We won't go over the details here (there are nice tutorials online), but it comes out in the form:

$$\frac{Y}{X} = \frac{C_0}{1-p_0\mathcal{R}} + \frac{C_1}{1-p_1\mathcal{R}} + \frac{C_2}{1-p_2\mathcal{R}} + \cdots + D_0 + D_1\mathcal{R} + D_2\mathcal{R}^2 + \cdots$$

where the C_k and D_k are constants defined in terms of the a_i and b_j from the original polynomial ratio. It's actually a little bit trickier than this: if there are complex poles, then for each conjugate pair of complex poles, we would put in a second-order system with real coefficients that expresses the contribution of the sum of the complex modes.

The constant term D_0 and the terms with $\mathcal{R}^k$ in the numerator occur if the numerator has equal or higher order than the denominator. They do not involve feedback and don't affect the long-term behavior of the system. One mode of the form p_i^n arises from each factor of the denominator. This modal decomposition leads us to an alternative block diagram:



We can fairly easily observe that the behavior is going to be the sum of the behaviors of the individual modes, and that, as in the second-order case, the mode whose pole has the largest magnitude will govern the qualitative long-term nature of the behavior of the system in response to a unit-sample input.

5.5.4 Finding Poles

In general, we will find that if the denominator of the system function H is a kth order polynomial, then it can be factored into the form $(1-p_0\mathcal{R})(1-p_1\mathcal{R})\ldots(1-p_{k-1}\mathcal{R})$. We will call the p_i values the *poles* of the system. The entire persistent output of the system can be expressed as a scaled sum of the signals arising from each of these individual poles.

We're doing something interesting here! We are using the PCAP system backwards for analysis. We have a complex thing that is hard to understand monolithically, so we are taking it apart into simpler pieces that we do understand.

It might seem like factoring polynomials in this way is tricky, but there is a straightforward way to find the poles of a system given its denominator polynomial in $\mathcal{R}$.

We'll start with an example. Assume the denominator is $12\mathcal{R}^2 - 7\mathcal{R} + 1$. If we play a quick trick, and introduce a new variable $z = 1/\mathcal{R}$, then our denominator becomes

$$\frac{12}{z^2} - \frac{7}{z} + 1 \ .$$

We'd like to find the roots of this polynomial, which, if you multiply through by z^2, is equivalent to finding the roots of this poyinomial:

$$12 - 7z + z^2 = 0 \ .$$

The roots are 3 and 4. If we go back to our original polynomial in $\mathcal{R}$, we can see that:

$$12\mathcal{R}^2 - 7\mathcal{R} + 1 = (1 - 4\mathcal{R})(1 - 3\mathcal{R}) \ .$$

so that our poles are 4 and 3. So, remember, *the poles* **are not** *the roots of the polynomial in $\mathcal{R}$, but* **are** *the roots of the polynomial in the reciprocal of $\mathcal{R}$.*

The roots of a polynomial can be a combination of real and complex numbers, with the requirement that if a complex number p is a root, then so is its complex conjugate, p^*.

Repeated roots

In some cases, the equation in z will have repeated roots. For example, $z^2 - z + 0.25$, which has two roots at 0.5. In this case, the system has a repeated pole; it is still possible to perform an additive decomposition, but it is somewhat trickier. Ultimately, however, it is still the magnitude of the largest root that governs the long-term convergence properties of the system.

5.5.5 Superposition

The principle of superposition states that the response of a LTI system to a sum of input signals is the sum of the responses of that system to the components of the input. So, given a system with system function H, and input $X = X_1 + X_2$,

$$Y = HX = H(X_1 + X_2) = HX_1 + HX_2$$

So, although we have been concentrating on the unit sample response of systems, we can see that, to find the response of a system to any finite signal, we must simply sum the responses to each of the components of that signal; and those responses will simply be scaled, delayed copies of the response to the unit sample.

If Φ is a polynomial in $\mathcal{R}$ and $X = \Phi\Delta$, then we can use what we know about the algebra of polynomials in $\mathcal{R}$ (remembering that H is a ratio of polynomials in $\mathcal{R}$) to determine that

$$Y = HX = H(\Phi\Delta) = \Phi(H\Delta)$$

So, for example, if $X = (-3\mathcal{R}^2 + 20\mathcal{R}^4)\Delta$, then $Y = HX = H(-3\mathcal{R}^2 + 20\mathcal{R}^4)\Delta = (-3\mathcal{R}^2 + 20\mathcal{R}^4)H\Delta$. **From this equation, it is easy to see that once we understand**

the unit-sample response of a system, we can see how it will respond to any finite input.

We might be interested in understanding how a system H responds to a *step* input signal. Let's just consider the basic step signal, U, defined as

$$u[n] = \begin{cases} 1 & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases} .$$

We can express U as an infinite sum of increasingly delayed unit-sample signals:

$$\begin{aligned} U &= \Delta + \mathcal{R}\Delta + \mathcal{R}^2\Delta + \mathcal{R}^3\Delta + \cdots \\ &= (1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots)\Delta \ . \end{aligned}$$

The response of a system to U will therefore be an infinite sum of unit-sample responses. Let $Z = H\Delta$ be the unit-sample response of H. Then

$$\begin{aligned} HU &= H(1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots)\Delta \\ &= (1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots)H\Delta \\ &= (1 + \mathcal{R} + \mathcal{R}^2 + \mathcal{R}^3 + \cdots)Z \end{aligned}$$

Let's consider the case where H is a first-order system with a pole at p. Then,

$$z[n] = p^n$$

If $Y = HU$, then

$$\begin{aligned} y[n] &= z[n] + z[n-1] + z[n-2] + \cdots + z[0] \\ &= \sum_{k=0}^{n} z[k] \\ &= \sum_{k=0}^{n} p^k \end{aligned}$$

It's clear that, if $|p| >= 1$ then $y[n]$ will grow without bound; but if $0 < p < 1$ then, as n goes to infinity, $y[n]$ will converge to $1/(1-p)$.

We won't study this in any further detail, but it's useful to understand that the basis of our analysis of systems applies very broadly across LTI systems and inputs.

5.6 Summary of System Behavior

Here is some terminology that will help us characterize the long-term behavior of systems.

- A signal is *transient* if it has finitely many non-zero samples.
- Otherwise, it is *persistent*.

- A signal is *bounded* if there is exist upper and lower bound values such that the samples of the signal never exceed those bounds; item otherwise it is *unbounded*.

Now, using those terms, here is what we can say about system behavior.

- A transient input to an acyclic (feed-forward) system results in a transient output.
- A transient input to a cyclic (feed-back) system results in a persistent output.
- The poles of a system are the roots of the denominator polynomial of the system function in $1/\mathcal{R}$.
- The dominant pole is the pole with the largest magnitude.
- If the dominant pole has magnitude > 1, then in response to a bounded input, the output signal will be unbounded.
- If the dominant pole has magnitude < 1, then in response to a bounded input, the output signal will be bounded; in response to a transient input, the output signal will converge to 0.
- If the dominant pole has magnitude 1, then in response to a bounded input, the output signal will be bounded; in response to a transient input, it will converge to some constant value.
- If the dominant pole is real and positive, then in response to a transient input, the signal will, after finitely many steps, begin to increase or decrease monotonically.
- If the dominant pole is real and negative, then in response to a transient input, the signal will, after finitely many steps, begin to alternate signs.
- If the dominant pole is complex, then in response to a transient input, the signal will, after finitely many steps, begin to be periodic, with a period of $2\pi/\Omega$, where Ω is the 'angle' of the pole.

5.7 Worked Examples

5.7.1 Specifying Difference Equations

Here are some examples of LTI systems and the way they would be described as difference equations. It's useful to pay careful attention to the specification of the coefficients. As a reminder, here's the general form.

$$\begin{aligned} y[n] &= c_0\, y[n-1] + c_1\, y[n-2] + \ldots + c_{k-1}\, y[n-k] \\ &\quad + d_0\, x[n] + d_1\, x[n-1] + \ldots + d_j\, x[n-j] \end{aligned}$$

- Output at step n is 3 times the input at step n:

$$y[n] = 3x[n]$$

dCoeffs: 3, cCoeffs: none

- Output at step n is 2 times the input at step $n-2$:

$$y[n] = 2x[n-2]$$

dCoeffs: 0, 0, 2, cCoeffs: none

- Output at step n is the input at step $n-1$ plus the output at step $n-2$:

$$y[n] = x[n-1] + y[n-2]$$

dCoeffs: 0, 1, cCoeffs: 0, 1

5.7.2 Difference Equations and Block Diagrams

Let H represent a system whose input is a signal X and whose output is a signal Y. The system H is defined by the following difference equations:

$$\begin{aligned} y[n] &= x[n] + z[n] \\ z[n] &= y[n-1] + z[n-1] \end{aligned}$$

Part A. Write down the operator equation of the sytem.

$$\begin{aligned} Y &= X + Z \\ Z &= \mathcal{R}Y + \mathcal{R}Z \end{aligned}$$

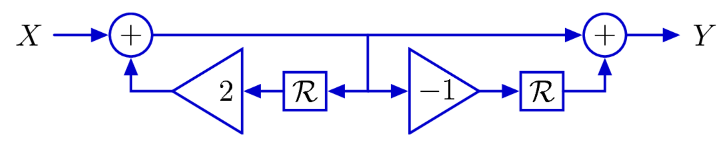
Now, we use the second equation to find an expression for Z:

$$Z = \frac{\mathcal{R}Y}{1-\mathcal{R}}$$

and substitute that into the first equation, and solve for Y:

$$\begin{aligned} Y &= X + \frac{\mathcal{R}Y}{1-\mathcal{R}} \\ (1-\mathcal{R})Y &= (1-\mathcal{R})X + \mathcal{R}Y \\ (1-2\mathcal{R})Y &= (1-\mathcal{R})X \\ Y &= \frac{1-\mathcal{R}}{1-2\mathcal{R}}X \end{aligned}$$

Part B. Which of the following systems are valid representations of H? (Remember that there can be multiple "equivalent" representations for a system.)



Equivalent to H (yes/no)? Ans: **YES**

Find operator equations here; start by naming the signal that's flowing between the two adders. Let's call it W.

$$\begin{aligned} Y &= W - \mathcal{R}W \\ W &= X + 2\mathcal{R}W \end{aligned}$$

Now, rewrite the first equation as

$$Y = (1 - \mathcal{R})W$$

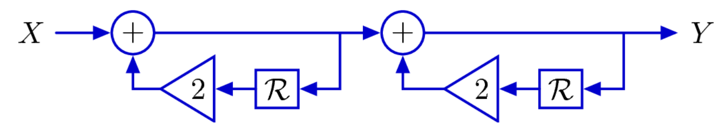
And the second one as

$$W = \frac{X}{1 - 2\mathcal{R}}$$

Then, combine them to get

$$\begin{aligned} Y &= (1 - \mathcal{R})\frac{X}{1 - 2\mathcal{R}} \\ &= \frac{1 - \mathcal{R}}{1 - 2\mathcal{R}}X \end{aligned}$$

Showing that this system is the same as H.



'Equivalent to H (yes/no)? Ans: **NO**

This time, let's name the signal that's flowing between the two adders A. Now, we have equations

$$\begin{aligned} Y &= A + 2\mathcal{R}Y \\ A &= X + 2\mathcal{R}A \end{aligned}$$

Rewrite the first equation as:

$$Y = \frac{A}{1 - 2\mathcal{R}}$$

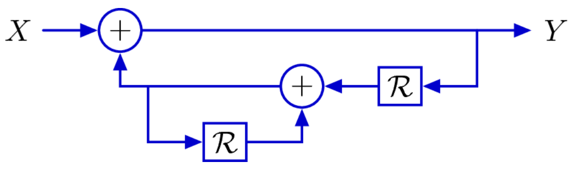
And the second as:

$$A = \frac{X}{1 - 2\mathcal{R}}$$

When we combine them, we get

$$Y = \frac{X}{(1-2\mathcal{R})(1-2\mathcal{R})}$$

which is not equivalent to the original system.



Equivalent to H (yes/no)? Ans: **YES**

We'll name the signal flowing between the adders B. We get equations

$$\begin{aligned} Y &= X + B \\ B &= \mathcal{R}Y + \mathcal{R}B \end{aligned}$$

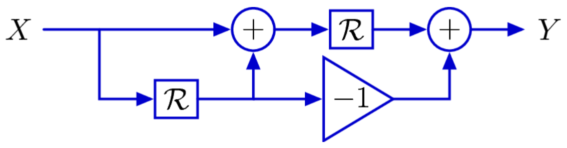
Rewriting the second equation, we have

$$B = \frac{\mathcal{R}Y}{1-\mathcal{R}}$$

Substituting into the first equation and solving, we get:

$$\begin{aligned} Y &= X + \frac{\mathcal{R}Y}{1-\mathcal{R}} \\ Y(1-\mathcal{R}) &= (1-\mathcal{R})X + \mathcal{R}Y \\ Y(1-2\mathcal{R}) &= (1-\mathcal{R})X \\ Y &= \frac{1-\mathcal{R}}{1-2\mathcal{R}}X \end{aligned}$$

So this system is equivalent to the original one.



Equivalent to H (yes/no)? Ans: **NO**

Let's call the signal coming out of the first adder C. We get equations

$$\begin{aligned} Y &= \mathcal{R}C - \mathcal{R}X \\ C &= X + \mathcal{R}X \end{aligned}$$

So

$$\begin{aligned} Y &= \mathcal{R}(X + \mathcal{R}X) - \mathcal{R}X \\ &= \mathcal{R}^2 X \end{aligned}$$

which is not equivalent to the original system.

Part C. Assume that the system starts “at rest” and that the input signal X is the unit sample signal. Determine $y[3]$.

Ans: $y[3] = 4$

n	$x[n]$	$z[n]$	$y[n]$
-1	0	0	0
0	1	0	1
1	0	1	1
2	0	2	2
3	0	4	4

Part D. Let p_o represent the dominant pole of H. Determine p_o. Answer p_0 or **none** if there are no poles:

Ans: 2

The denominator polynomial of the system function is $1 - 2\mathcal{R}$. This is directly in the form that exposes the pole as 2. But, we can also go step by step. We convert this into a polynomial in $z = 1/\mathcal{R}$ to get $1 - 2/z$. The roots of that equation are the same as the roots of $z - 2 = 0$; the single root is 2.

5.7.3 Cool Difference Equations

Newton’s law of cooling states that: the rate of change of the temperature of an object is proportional to the difference between its own temperature and the temperature of its surroundings.

We can model this process in discrete time, by assuming that the change in an object’s temperature from one time step to the next is proportional to the difference (on the earlier step) between the temperature of the object and the temperature of the environment, as well as to the length of the time step.

Let

- $o[n]$ be temperature of object
- $s[n]$ be temperature of environment
- T be the duration of a time step
- K be the constant of proportionality

Part A. Write a difference equation for Newton's law of cooling. Be sure the signs are such that the temperature of the object will eventually equilibrate with that of the environment.

Ans: $o[n] = o[n-1] + TK(s[n-1] - o[n-1])$

Part B. Write the system function corresponding to this equation (show your work):

Ans: $H = \frac{O}{S} = \frac{KT\mathcal{R}}{1-(1-KT)\mathcal{R}}$

First, convert the difference equation to an operator equation, then solve for O in terms of S.

$$\begin{aligned} O &= \mathcal{R}O + KT(\mathcal{R}S - \mathcal{R}O) \\ O - \mathcal{R}O + KT\mathcal{R}O &= KT\mathcal{R}S \\ O(1-(1-KT)\mathcal{R}) &= KT\mathcal{R}S \\ \frac{O}{S} &= \frac{KT\mathcal{R}}{1-(1-KT)\mathcal{R}} \end{aligned}$$

5.7.4 On the Verge

For each difference equation below, say whether, for a unit sample input signal:

- the output of the system it describes will diverge or not,
- the output of the system it describes (a) will always be positive, (b) will alternate between positive and negative, or (c) will have a different pattern of oscillation

1. $10y[n] - y[n-1] = 8x[n-3]$

   Does the above diverge? Ans: No

   Does the above alternate, oscillate, or remain positive? Ans: Positive

   We first write the operator equation:

   $$10Y - \mathcal{R}Y = 8\mathcal{R}^3X$$

   And the system function

   $$\frac{Y}{X} = \frac{8\mathcal{R}^3}{10-\mathcal{R}}$$

   Find the root of the polynomial in $z = 1/\mathcal{R}$:

   $$\begin{aligned} 10z - 1 &= 0 \\ z &= 0.1 \end{aligned}$$

   The single pole is at 0.1. It is positive, so for a unit-sample input, the output will always be positive (assuming it starts at rest). It has magnitude less than 1, so it will converge.

2. $y[n] = -y[n-1] - 10y[n-2] + x[n]$

   Does the above diverge? Ans: Yes

   Does the above alternate, oscillate, or remain positive? Ans: Oscillates

   We first write the operator equation:

$$Y + \mathcal{R}Y + 10\mathcal{R}^2 Y = X$$

   And the system function
$$\frac{Y}{X} = \frac{1}{1 + \mathcal{R} + 10\mathcal{R}^2}$$
   Find the roots of the polynomial in $z = 1/\mathcal{R}$:

$$\begin{aligned} Z^2 + Z + 10 &= 0 \\ Z &= \frac{-1 \pm \sqrt{1 - 100}}{2} \\ Z &= 0.5 \pm 4.97j \end{aligned}$$

   The magnitude of the poles is 5, which is greater than 1, so it diverges. The poles are complex, so the output will oscillate.

3. $y[n] = -0.6y[n-1] + .16y[n-2] - 0.1x[n-1]$

   Does the above diverge? Ans: No

   Does the above alternate, oscillate, or remain positive? Ans: Oscillates

   We first write the operator equation:

$$Y + 0.6\mathcal{R}Y - .16\mathcal{R}^2 Y = -0.1\mathcal{R}X$$

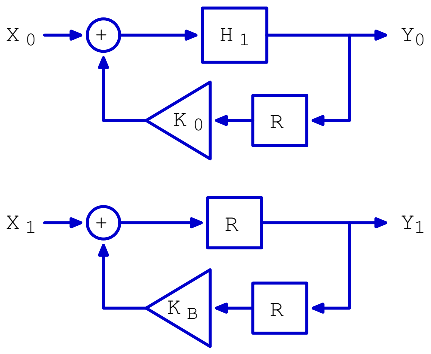
   And the system function
$$\frac{Y}{X} = \frac{-0.1\mathcal{R}}{1 + 0.6\mathcal{R} - 0.16\mathcal{R}^2}$$
   Find the roots of the polynomial in $z = 1/\mathcal{R}$:

$$\begin{aligned} Z^2 + 0.6Z - 0.16 &= 0 \\ Z &= \frac{-0.6 \pm \sqrt{.36 + .64}}{2} \\ Z &= (-0.8, 0.2) \end{aligned} \tag{5.4}$$

   The dominant pole is -0.8, because it has the largest magnitude. Its magnitude is less than 1, so the system will converge. The pole is negative, so the system will alternate positive and negative signs.

5.7.5 System functions

Let H_1 represent a subsystem that is part of the larger system shown below.



The system function for the larger system can be written as

$$H_0 = \frac{Y_0}{X_0} = \frac{H_1}{1 - K_0 R H_1}\,.$$

Assume that $H_1 = H_{1B} = \frac{Y_1}{X_1}$ as shown below.

What is the system function for H_{1B}?

Ans: $H_{1B} = \dfrac{\mathcal{R}}{1 - K_B \mathcal{R}^2}$

Determine the system function H_0 for the larger system when $H_1 = H_{1B}$.

Ans: $H_0 = \dfrac{\mathcal{R}}{1 + (K_0 - K_B)R^2}$

Explain under what conditions on K_0 and K_B is this system stable?

Ans: There are poles at $z = \pm\sqrt{K_B - K_0}$. To be stable, the poles should all have magnitude less than 1. Thus the system is stable if $|K_B - K_0| < 1$.

Explain under what conditions on K_0 and K_B does the unit-sample response decay monotonically?

Ans: None. For monotonic convergence, **both** poles must have magnitudes between 0 and 1 (since there are two poles of equal magnitude). If $K_B < K_0$ then the poles have non-zero imaginary parts, and the response oscillates. If $K_B > K_0$ then one pole is on the positive real axis and one is on the negative real axis. The pole on the negative real axis causes the unit sample response to alternate. Thus there are no values of K_O and K_B for which there is monotonic decay.

Chapter 6

Long-Term Decision-Making and Search

In the lab exercises of this course, we have implemented several brains for our robots. We used wall-following to navigate through the world, and we used various linear controllers to drive down the hall. In the first case, we just wrote a program that we hoped would do a good job. When we were studying linear controllers, we, as designers, made models of the controller's behavior in the world and tried to prove whether it would behave in the way we wanted it to, taking into account a longer-term pattern of behavior.

Often, we will want a system to generate complex long-term patterns of behavior, but we will not be able to write a simple control rule to generate those behavior patterns. In that case, we'd like the system to evaluate alternatives for itself, but instead of evaluating single actions, it will have to evaluate whole sequences of actions, deciding whether they're a good thing to do given the current state of the world.

Let's think of the problem of navigating through a city, given a road map, and knowing where we are. We can't usually decide whether to turn left at the next intersection without deciding on a whole path.

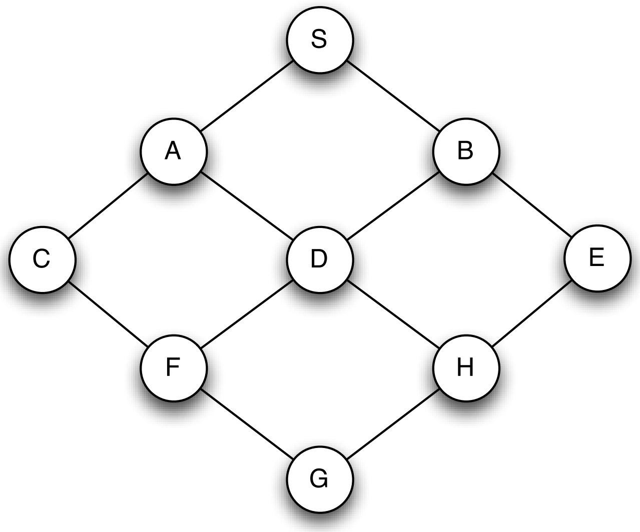
As always, the first step in the process will be to come up with a formal model of a real-world problem that abstracts away the irrelevant detail. So, what, exactly, is a path? The car we're driving will actually follow a trajectory through continuous space(time), but if we tried to plan at that level of detail we would fail miserably. Why? First, because the space of possible trajectories through two-dimensional space is just too enormous. Second, because when we're trying to decide which roads to take, we don't have the information about where the other cars will be on those roads, which will end up having a huge effect on the detailed trajectory we'll end up taking.

So, we can divide the problem into two levels: planning in advance which turns we'll make at which intersections, but deciding 'on-line', while we're driving, exactly how to control the steering wheel and the gas to best move from intersection to intersection, given the current circumstances (other cars, stop-lights, etc.).

We can make an abstraction of the driving problem to include road intersections and the way they're connected by roads. Then, given a start and a goal intersection, we could consider all possible paths between them, and choose the one that is best.

What criteria might we use to evaluate a path? There are all sorts of reasonable ones: distance, time, gas mileage, traffic-related frustration, scenery, etc. Generally speaking, the approach we'll outline below can be extended to any criterion that is additive: that is, your happiness with the whole path is the sum of your happiness with each of the segments. We'll start with the simple criterion of wanting to find a path with the fewest "steps"; in this case, it will be the path that traverses the fewest intersections. Then in Section 6.5 we will generalize our methods to handle problems where different actions have different costs.

One possible algorithm for deciding on the best path through a map of the road intersections in this (very small) world



would be to enumerate all the paths, evaluate each one according to our criterion, and then return the best one. The problem is that there are *lots* of paths. Even in our little domain, with 9 intersections, there are 210 paths from the intersection labeled S to the one labeled G.

We can get a much better handle on this problem, by formulating it as an instance of a graph search (or a "state-space search") problem, for which there are simple algorithms that perform well.

6.1 State-Space Search

We'll model a state-space search problem formally as

- a (possibly infinite) set of *states* the system can be in;
- a *starting state*, which is an element of the set of states;
- a *goal test*, which is a procedure that can be applied to any state, and returns `True` if that state can serve as a goal;[1]

[1]Although in many cases we have a particular goal state (such as the intersection in front of my

- a *successor function*, which takes a state and an action as input, and returns the new state that will result from taking the action in the state; and
- a *legal action list*, which is just a list of actions that can be legally executed in this domain.

The decision about what constitutes an *action* is a modeling decision. It could be to drive to the next intersection, or to drive a meter, or a variety of other things, depending on the domain. The only requirement is that it terminates in a well-defined next state (and that, when it is time to execute the plan, we will know how to execute the action).

We can think of this model as specifying a *labeled graph* (in the computer scientist's sense), in which the states are the nodes, action specifies which of the arcs leading out of a node is to be selected, and the successor function specifies the node at the end of each of the arcs.

So, for the little world above, we might make a model in which

- The set of states is the intersections `'S'`, `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, `'F'`, `'G'`, `'H'`.
- The starting state is `'S'`.
- The goal test is something like:

  ```
  lambda x: x == 'H'
  ```

- The legal actions in this domain are the numbers `0`, `1`, ..., `n-1`, where `n` is the maximum number of successors in any of the states (see map below).
- The map can be defined using a dictionary:

  ```
  map1 = {'S' : ['A', 'B'],
          'A' : ['S', 'C', 'D'],
          'B' : ['S', 'D', 'E'],
          'C' : ['A', 'F'],
          'D' : ['A', 'B', 'F', 'H'],
          'E' : ['B', 'H'],
          'F' : ['C', 'D', 'G'],
          'H' : ['D', 'E', 'G'],
          'G' : ['F', 'H']}
  ```

  where each key is a state, and its value is a list of states that can be reached from it in one step.

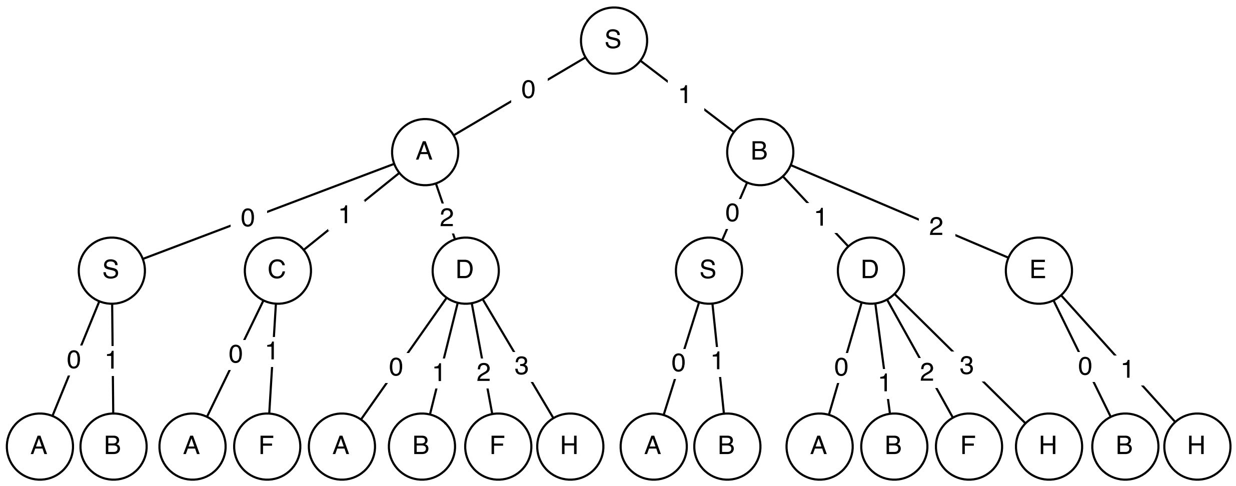
  Now we can define the successor function as

  ```
  def map1successors(s, a):
      return map1[s][a]
  ```

house), in other cases, we may have the goal of going to any gas station, which can be satisfied by many different intersections.

but with an additional test to be sure that if we attempt to take an action that doesn't exist in `s`, it just results in state `s`. So, for example, the successor reached from state `'A'` by taking action `1` is state `'C'`.

We can think of this structure as defining a *search tree*, like this:



It has the starting state, S, at the root node, and then each node has its successor states as children. Layer k of this tree contains all possible paths of length k through the graph.

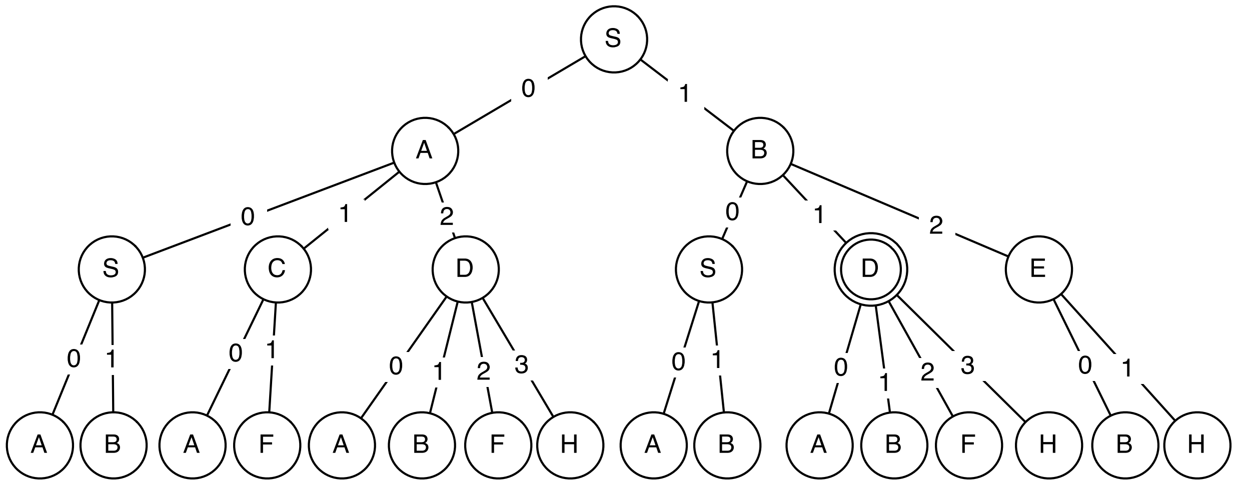
6.1.1 Representing Search Trees

We will need a way to represent the tree as a Python data structure as we construct it during the search process. We will start by defining a class to represent a *search node*, which is one of the circles in the tree.

Each search node represents:

- The state of the node;
- the action that was taken to arrive at the node; and
- the search node *from which* this node can be reached.

We will call the node from which a node can be reached its *parent* node. So, for example, in the figure below



we will represent the node with double circles around it with its state, `'D'`, the action that reached it, `1`, and its parent node, which is the node labeled `'B'` above it.

Note that *states and nodes are not the same thing!* In this tree, there are many nodes labeled by the same state; they represent different paths to and through the state.

Here is a Python class representing a search node. It's pretty straightforward.

```
class SearchNode:
    def __init__(self, action, state, parent):
        self.state = state
        self.action = action
        self.parent = parent
```

There are a couple of other useful methods for this class. First, the `path` method, returns a list of pairs `(a, s)` corresponding to the path starting at the top (root) of the tree, going down to this node. It works its way up the tree, until it reaches a node whose parent is `None`.

```
    def path(self):
        if self.parent == None:
            return [(self.action, self.state)]
        else:
            return self.parent.path() + [(self.action, self.state)]
```

The path corresponding to our double-circled node is `((None, 'S'), (1, 'B'), (1, 'D'))`.

Another helper method that we will find useful is the `inPath` method, which takes a state, and returns `True` if the state occurs anywhere in the path from the root to the node.

```
    def inPath(self, s):
        if s == self.state:
            return True
        elif self.parent == None:
            return False
        else:
            return self.parent.inPath(s)
```

6.1.2 Basic Search Algorithm

We'll describe a sequence of search algorithms of increasing sophistication and efficiency. An ideal algorithm will take a problem description as input and return a path from the start to a goal state, if one exists, and return None, if it does not. Some algorithms will not be capable of finding a path in all cases.

How can we systematically search for a path to the goal? There are two plausible strategies:

- Start down a path, keep trying to extend it until you get stuck, in which case, go back to the last choice you had, and go a different way. This is how kids often solve mazes. We'll call it *depth-first* search.
- Go layer by layer through the tree, first considering all paths of length 1, then all of length 2, etc. We'll call this *breadth-first* search.

Both of the search strategies described above can be implemented using a procedure with this basic structure:

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = EmptyAgenda()
    add(SearchNode(None, initialState, None), agenda)
    while not empty(agenda):
        parent = getElement(agenda)
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            else:
                add(newN, agenda)
    return None
```

We start by checking to see if the initial state is a goal state. If so, we just return a path consisting of the initial state.

Otherwise, we have real work to do. We make the *root* node of the tree. It has no parent, and there was no action leading to it, so all it needs to specify is its state, which is `initialState`, so it is created with

```
SearchNode(None, initialState, None)
```

During the process of constructing the search tree, we will use a data structure, called an *agenda*, to keep track of which nodes in the partially-constructed tree are on the fringe, ready to be expanded, by adding their children to the tree. We initialize the agenda to contain the root node. Now, we enter a loop that will run until the agenda is empty (we have no more paths to consider), but could stop sooner.

Inside the loop, we select a node from the agenda (more on how we decide which one to take out in a bit) and *expand it*. To expand a node, we determine which actions can be taken from the state that is stored in the node, and *visit* the *successor* states that can be reached via the actions.

When we visit a state, we make a new search node (`newN`, in the code) that has the node we are in the process of expanding as the parent, and that remembers the state being visited and the action that brought us here from the parent.

Next, we check to see if the new state satisfies the goal test. If it does, we're done! We return the path associated with the new node.

If this state doesn't satisfy the goal test, then we add the new node to the agenda. We continue this process until we find a goal state or the agenda becomes empty. This is not quite yet an algorithm, though, because we haven't said anything about what it means to add and extract nodes from the agenda. And we'll find that it will do some very stupid things in its current form.

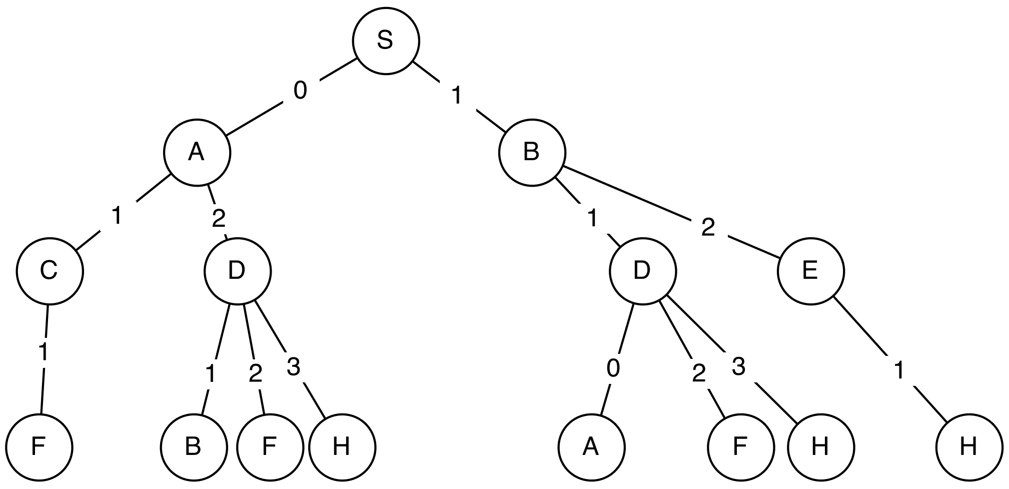
We'll start by curing the stupidities, and then return to the question of how best to select nodes from the agenda.

6.1.3 Basic Pruning or How Not to be Completely Stupid

If you examine the full search tree, you can see that some of the paths it contains are completely ridiculous. It can never be reasonable, if we're trying to find the shortest path between two states, to go back to a state we have previously visited on that same path. So, to avoid trivial infinite loops, we can adopt the following rule:

Pruning Rule 1: Don't consider any path that visits the same state twice.

If we can apply this rule, then we will be able to remove a number of branches from the tree, as shown here:



It is relatively straightforward to modify our code to implement this rule:

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = [SearchNode(None, initialState, None)]
    while agenda != []:
        parent = getElement(agenda)
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif parent.inPath(newS):
                pass
            else:
                add(newN, agenda)
    return None
```

The red text above shows the code we've added to our basic algorithm. It just checks to see whether the current state already exists on the path to the node we're expanding and, if so, it doesn't do anything with it.

The next pruning rule doesn't make a difference in the current domain, but can have a big effect in other domains:

Pruning Rule 2: If there are multiple actions that lead from a state r to a state s, consider only one of them.

To handle this in the code, we have to keep track of which new states we have reached in expanding this node, and if we find another way to reach one of those states, we just ignore it. The changes to the code for implementing this rule are shown in red:

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = [SearchNode(None, initialState, None)]
    while agenda != []:
        parent = getElement(agenda)
        newChildStates = []
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:
                pass
            elif parent.inPath(newS):
                pass
            else:
                newChildStates.append(newS)
                add(newN, agenda)
    return None
```

Each time we pick a new node to expand, we make a new empty list, `newChildStates`, and keep track of all of the new states we have reached from this node.

Now, we have to think about how to extract nodes from the agenda.

6.1.4 Stacks and Queues

In designing algorithms, we frequently make use of two simple data structures: stacks and queues. You can think of them both as abstract data types that support two operations: `push` and `pop`. The `push` operation adds an element to the stack or queue, and the `pop` operation removes an element. The difference between a stack and a queue is what element you get back when you do a `pop`.

- **stack**: When you `pop` a stack, you get back the element that you most recently put in. A stack is also called a LIFO, for *last in, first out*.

- **queue**: When you `pop` a queue, you get back the element that you put in earliest. A queue is also called a FIFO, for *first in, first out*.

In Python, we can use lists to represent both stacks and queues. If `data` is a list, then `data.pop(0)` removes the first element from the list and returns it, and `data.pop()` removes the last element and returns it.

Here is a class representing stacks as lists. It always adds new elements to the end of the list, and pops items off of the same end, ensuring that the most recent items get popped off first.

```
class Stack:
    def __init__(self):
        self.data = []
    def push(self, item):
        self.data.append(item)
    def pop(self):
        return self.data.pop()
    def isEmpty(self):
        return self.data is []
```

Here is a class representing stacks as lists. It always adds new elements to the end of the list, and pops items off of the front, ensuring that the oldest items get popped off first.

```
class Queue:
    def __init__(self):
        self.data = []
    def push(self, item):
        self.data.append(item)
    def pop(self):
        return self.data.pop(0)
    def isEmpty(self):
        return self.data is []
```

We will use stacks and queues to implement our search algorithms.

6.1.5 Depth-First Search

Now we can easily describe depth-first search by saying that it's an instance of the generic search procedure described above, but in which the agenda is a *stack*: that is, we always expand the node we most recently put into the agenda.

The code listing below shows our implementation of depth-first search.

```
def depthFirstSearch(initialState, goalTest, actions, successor):
    agenda = Stack()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.isEmpty():
        parent = agenda.pop()
        newChildStates = []
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:
                pass
            elif parent.inPath(newS):
                pass
            else:
                newChildStates.append(newS)
                agenda.push(newN)
    return None
```

You can see several operations on the agenda. We:

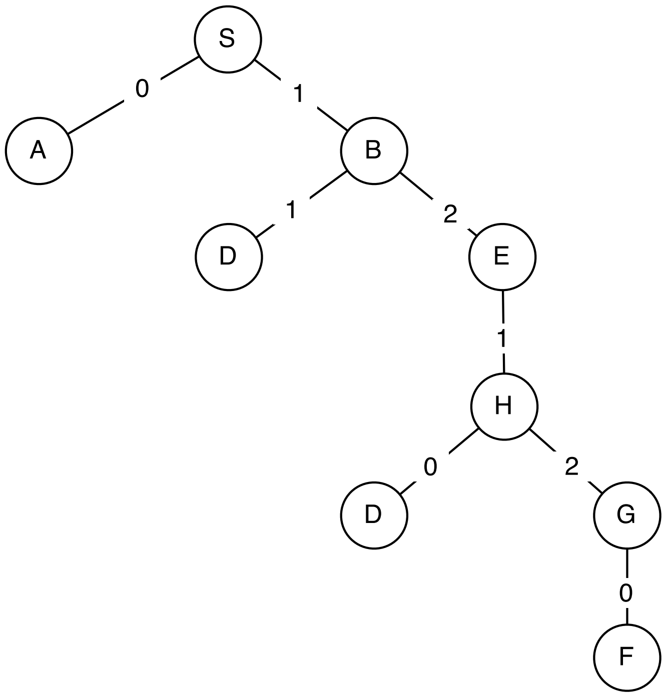
- Create an empty `Stack` instance, and let that be the agenda.
- Push the initial node onto the agenda.
- Test to see if the agenda is empty.
- Pop the node to be expanded off of the agenda.
- Push newly visited nodes onto the agenda.

Because the agenda is an instance of the `Stack` class, subsequent operations on the agenda ensure that it will act like a stack, and guarantee that children of the most recently expanded node will be chosen for expansion next.

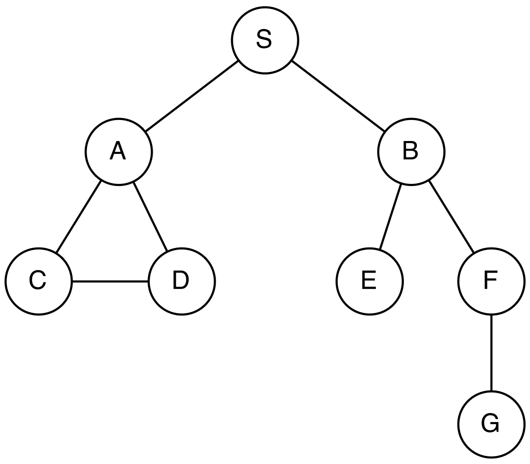
So, let's see how this search method behaves on our city map, with start state S and goal state F. Here is a trace of the algorithm (you can get this in the code we distribute by setting `verbose = True` before you run it.)

```
depthFirst('S', lambda x: x == 'F', map1LegalActions, map1successors)
agenda:  Stack([S])
   expanding:  S
agenda:  Stack([S-0->A, S-1->B])
   expanding:  S-1->B
agenda:  Stack([S-0->A, S-1->B-1->D, S-1->B-2->E])
   expanding:  S-1->B-2->E
agenda:  Stack([S-0->A, S-1->B-1->D, S-1->B-2->E-1->H])
   expanding:  S-1->B-2->E-1->H
agenda:  Stack([S-0->A, S-1->B-1->D, S-1->B-2->E-1->H-0->D, S-1->B-2->E-1->H-2->G])
   expanding:  S-1->B-2->E-1->H-2->G
8  states visited
[(None, 'S'), (1, 'B'), (2, 'E'), (1, 'H'), (2, 'G'), (0, 'F')]
```

You can see that in this world, the search never needs to "backtrack", that is, to go back and try expanding an older path on its agenda. It is always able to push the current path forward until it reaches the goal. Here is the search tree generated during the depth-first search process.



Here is another city map:



In this city, depth-first search behaves a bit differently (trying to go from S to D this time):

```
depthFirst('S', lambda x: x == 'D', map2LegalActions, map2successors)
agenda:  Stack([S])
   expanding:  S
agenda:  Stack([S-0->A, S-1->B])
   expanding:  S-1->B
agenda:  Stack([S-0->A, S-1->B-1->E, S-1->B-2->F])
   expanding:  S-1->B-2->F
agenda:  Stack([S-0->A, S-1->B-1->E, S-1->B-2->F-1->G])
   expanding:  S-1->B-2->F-1->G
agenda:  Stack([S-0->A, S-1->B-1->E])
   expanding:  S-1->B-1->E
agenda:  Stack([S-0->A])
   expanding:  S-0->A
7  states visited
[(None, 'S'), (0, 'A'), (2, 'D')]
```

In this case, it explores all possible paths down in the right branch of the world, and then has to backtrack up and over to the left branch.

Here are some important properties of depth-first search:

- It will run forever if we don't apply pruning rule 1, potentially going back and forth from one state to another, forever.
- It may run forever in an infinite domain (as long as the path it's on has a new successor that hasn't been previously visited, it can go down that path forever).
- It doesn't necessarily find the shortest path (as we can see from the very first example).

- It is generally efficient in the amount of space it requires to store the agenda, which will be a constant factor times the depth of the path it is currently considering (we'll explore this in more detail in Section 6.4.)

6.1.6 Breadth-First Search

To change to breadth-first search, we need to choose the oldest, rather than the newest paths from the agenda to expand. All we have to do is change the agenda to be a queue instead of a stack, and everything else stays the same, in the code.

```
def breadthFirstSearch(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.isEmpty():
        parent = agenda.pop()
        newChildStates = []
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:
                pass
            elif parent.inPath(newS):
                pass
            else:
                newChildStates.append(newS)
                agenda.push(newN)
    return None
```

Here is how breadth-first search works, looking for a path from S to F in our first city:

```
>>> breadthFirst('S', lambda x: x == 'F', map1LegalActions, map1successors)
agenda:  Queue([S])
   expanding:  S
agenda:  Queue([S-0->A, S-1->B])
   expanding:  S-0->A
agenda:  Queue([S-1->B, S-0->A-1->C, S-0->A-2->D])
   expanding:  S-1->B
agenda:  Queue([S-0->A-1->C, S-0->A-2->D, S-1->B-1->D, S-1->B-2->E])
   expanding:  S-0->A-1->C
7  states visited
[(None, 'S'), (0, 'A'), (1, 'C'), (1, 'F')]
```

We can see it proceeding systematically through paths of length two, then length three, finding the goal among the length-three paths.

Here are some important properties of breadth-first search:

- Always returns a shortest (least number of steps) path to a goal state, if a goal state exists in the set of states reachable from the start state.
- It may run forever if there is no solution and the domain is infinite.
- It requires more space than depth-first search.

6.1.7 Dynamic Programming

Let's look at breadth-first search in the first city map example, but this time with goal state G:

```
>>> breadthFirst('S', lambda x: x == 'G', map1LegalActions, map1successors)
agenda:  Queue([S])
   expanding:  S
agenda:  Queue([S-0->A, S-1->B])
   expanding:  S-0->A
agenda:  Queue([S-1->B, S-0->A-1->C, S-0->A-2->D])
   expanding:  S-1->B
agenda:  Queue([S-0->A-1->C, S-0->A-2->D, S-1->B-1->D, S-1->B-2->E])
   expanding:  S-0->A-1->C
agenda:  Queue([S-0->A-2->D, S-1->B-1->D, S-1->B-2->E, S-0->A-1->C-1->F])
   expanding:  S-0->A-2->D
agenda:  Queue([S-1->B-1->D, S-1->B-2->E, S-0->A-1->C-1->F, S-0->A-2->D-1->B,
S-0->A-2->D-2->F, S-0->A-2->D-3->H])
   expanding:  S-1->B-1->D
agenda:  Queue([S-1->B-2->E, S-0->A-1->C-1->F, S-0->A-2->D-1->B, S-0->A-2->D-2->F,
S-0->A-2->D-3->H, S-1->B-1->D-0->A, S-1->B-1->D-2->F, S-1->B-1->D-3->H])
   expanding:  S-1->B-2->E
agenda:  Queue([S-0->A-1->C-1->F, S-0->A-2->D-1->B, S-0->A-2->D-2->F, S-0->A-2->
D-3->H, S-1->B-1->D-0->A, S-1->B-1->D-2->F, S-1->B-1->D-3->H, S-1->B-2->E-1->H])
   expanding:  S-0->A-1->C-1->F
16  states visited
[(None, 'S'), (0, 'A'), (1, 'C'), (1, 'F'), (2, 'G')]
```

The first thing that is notable about this trace is that it ends up visiting 16 states in a domain with 9 different states. The issue is that it is exploring multiple paths to the same state. For instance, it has both `S-0->A-2->D` and `S-1->B-1->D` in the agenda. Even worse, it has both `S-0->A` and `S-1->B-1->D-0->A` in there! We really don't need to consider all of these paths. We can make use of the following example of the dynamic programming principle:

The shortest path from X to Z that goes through Y is made up of the shortest path from X to Y and the shortest path from Y to Z.

So, as long as we find the shortest path from the start state to some intermediate state, we don't need to consider any other paths between those two states; there is no way that they can be part of the shortest path between the start and the goal. This insight is the basis of a new pruning principle:

Pruning Rule 3: Don't consider any path that visits a state that you have already visited via some other path.

In breadth-first search, because of the orderliness of the expansion of the layers of the search tree, we can guarantee that the first time we visit a state, we do so along the shortest path. So, we'll keep track of the states that we have visited so far, by using a dictionary, called `visited` that has an entry for every state we have visited.[2] Then, if we are considering adding a new node to the tree that goes to a state we have already visited, we just ignore it. This test can take the place of the test we used to have for pruning rule 1; it's clear that if the path we are considering already contains this state, then the state has been visited before. Finally, we have to remember, whenever we add a node to the agenda, to add the corresponding state to the visited list.

Here is our breadth-first search code, modified to take advantage of dynamic programming.

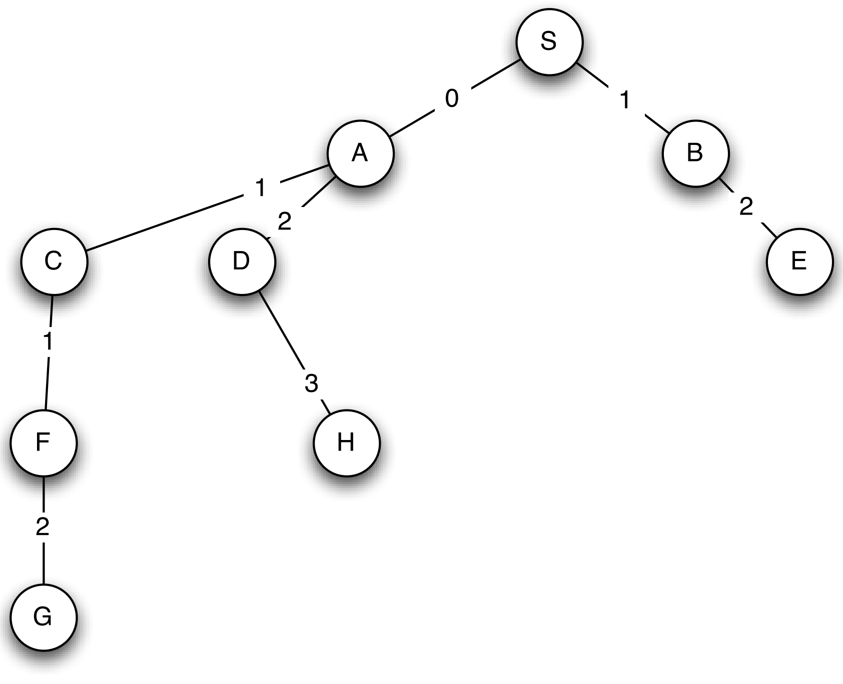
```
def breadthFirstDP(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    visited = {initialState: True}
    while not agenda.isEmpty():
        parent = agenda.pop()
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif visited.has_key(newS):
                pass
            else:
                visited[newS] = True:
                agenda.push(newN)
    return None
```

So, let's see how this performs on the task of going from S to G in the first city map:

[2]An alternative representation would be just to keep a Python `set` of visited nodes.

```
>>> breadthFirstDP('S', lambda x: x == 'G', map1LegalActions, map1successors)
agenda:  Queue([S])
   expanding:  S
agenda:  Queue([S-0->A, S-1->B])
   expanding:  S-0->A
agenda:  Queue([S-1->B, S-0->A-1->C, S-0->A-2->D])
   expanding:  S-1->B
agenda:  Queue([S-0->A-1->C, S-0->A-2->D, S-1->B-2->E])
   expanding:  S-0->A-1->C
agenda:  Queue([S-0->A-2->D, S-1->B-2->E, S-0->A-1->C-1->F])
   expanding:  S-0->A-2->D
agenda:  Queue([S-1->B-2->E, S-0->A-1->C-1->F, S-0->A-2->D-3->H])
   expanding:  S-1->B-2->E
agenda:  Queue([S-0->A-1->C-1->F, S-0->A-2->D-3->H])
   expanding:  S-0->A-1->C-1->F
8  states visited
[(None, 'S'), (0, 'A'), (1, 'C'), (1, 'F'), (2, 'G')]
```

As you can see, this results in visiting significantly fewer states. Here is the tree generated by this process:



In bigger problems, this effect will be amplified hugely, and will make the difference between whether the algorithm can run in a reasonable amount of time, and not.

We can make the same improvement to depth-first search; we just need to use a stack instead of a queue in the algorithm above. It still will not guarantee that the shortest path will be found, but will guarantee that we never visit more paths than the actual number of states.

6.1.8 Configurable Search Code

Because all of our search algorithms (breadth-first and depth-first, with and without dynamic programming) are all so similar, and we don't like to repeat code, we provide (in file `search.py`) a single, configurable search procedure. It also prints out some information as it goes, if you have the `verbose` or `somewhatVerbose` variables set to `True`, and has a limit on the maximum number of nodes it will expand (to keep from going into an infinite loop).

```
def search(initialState, goalTest, actions, successor,
           depthFirst = False, DP = True, maxNodes = 10000):
    if depthFirst:
        agenda = Stack()
    else:
        agenda = Queue()

    startNode = SearchNode(None, initialState, None)
    if goalTest(initialState):
        return startNode.path()
    agenda.push(startNode)
    if DP: visited = {initialState: True}
    count = 1
    while not agenda.isEmpty() and maxNodes > count:
        n = agenda.pop()
        newStates = []
        for a in actions:
            newS = successor(n.state, a)
            newN = SearchNode(a, newS, n)
            if goalTest(newS):
                return newN.path()
            elif newS in newStates:
                pass
            elif ((not DP) and n.inPath(newS)) or \
                  (DP and visited.has_key(newS)):
                pass
            else:
                count += 1
                if DP: visited[newS] = True
                newStates.append(newS)
                agenda.push(newN)
    return None
```

6.2 Connection to State Machines

We can use state machines as a convenient representation of state-space search problems. Given a state machine, in its initial state, what sequence of inputs can we feed to it to get it to enter a done state? This is a search problem, analogous to determining the sequence of actions that can be taken to reach a goal state.

The `getNextValues` method of a state machine can serve as the `successor` function in a search (the inputs to the machine are the actions). Our standard machines do not have a notion of legal actions; but we will add an attribute called `legalInputs`, which is a list of values that are legal inputs to the machine (these are the actions, from the planning perspective) to machines that we want to use with a search.

The `startState` attribute can serve as the initial state in the search and the `done` method of the machine can serve as the goal test function.

Then, we can plan a sequence of actions to go from the start state to one of the done states using this function, where `smToSearch` is an instance of `sm.SM`.

```
def smSearch(smToSearch, initialState = None, goalTest = None, maxNodes = 10000,
             depthFirst = False, DP = True):
   if initialState == None:
       initialState = smToSearch.startState
   if goalTest == None:
       goalTest = smToSearch.done
   return search(initialState, goalTest, smToSearch.legalInputs,
                 # This returns the next state
                 lambda s, a: smToSearch.getNextValues(s, a)[0],
                 maxNodes = maxNodes,
                 depthFirst=depthFirst, DP=DP)
```

It is mostly clerical: it allows us to specify a different initial state or goal test if we want to, and it extracts the appropriate functions out of the state machine and passes them into the search procedure. Also, because `getNextValues` returns both a state and an output, we have to wrap it inside a function that just selects out the next state and returns it.

6.3 Numeric Search Domain

Many different kinds of problems can be formulated in terms of finding the shortest path through a space of states. We'll explore a simple one here:

- The states are the integers.
- The initial state is some integer; let's say 1.

- The legal actions are to apply the following operations: $\{2n, n+1, n-1, n^2, -n\}$.
- The goal test is `lambda x:  x == 10` .

So, the idea would be to find a short sequence of operations to move from 1 to 10.

Here it is, formalized as state machine in Python:

```
class NumberTestSM(sm.SM):
    startState = 1
    legalInputs = ['x*2', 'x+1', 'x-1', 'x**2', '-x']
    def __init__(self, goal):
        self.goal = goal
    def nextState(self, state, action):
        if action == 'x*2':
            return state*2
        elif action == 'x+1':
            return state+1
        elif action == 'x-1':
            return state-1
        elif action == 'x**2':
            return state**2
        elif action == '-x':
            return -state
    def getNextValues(self, state, action):
        nextState = self.nextState(state, action)
        return (nextState, nextState)
    def done(self, state):
        return state == self.goal
```

First of all, this is a bad domain for applying depth-first search. Why? Because it will go off on a gigantic chain of doubling the starting state, and never find the goal. We can run breadth-first search, though. Without dynamic programming, here is what happens (we have set `verbose = False` and `somewhatVerbose = True` in the search file):

```
>>> smSearch(NumberTestSM(10), initialState = 1, depthFirst = False, DP = False)
   expanding:  1
   expanding:  1-x*2->2
   expanding:  1-x-1->0
   expanding:  1--x->-1
   expanding:  1-x*2->2-x*2->4
   expanding:  1-x*2->2-x+1->3
   expanding:  1-x*2->2--x->-2
   expanding:  1-x-1->0-x-1->-1
   expanding:  1--x->-1-x*2->-2
   expanding:  1--x->-1-x+1->0
   expanding:  1-x*2->2-x*2->4-x*2->8
   expanding:  1-x*2->2-x*2->4-x+1->5
33  states visited
[(None, 1), ('x*2', 2), ('x*2', 4), ('x+1', 5), ('x*2', 10)]
```

We find a nice short path, but visit 33 states. Let's try it with DP:

```
>>> smSearch(NumberTestSM(10), initialState = 1, depthFirst = False, DP = True)
   expanding:  1
   expanding:  1-x*2->2
   expanding:  1-x-1->0
   expanding:  1--x->-1
   expanding:  1-x*2->2-x*2->4
   expanding:  1-x*2->2-x+1->3
   expanding:  1-x*2->2--x->-2
   expanding:  1-x*2->2-x*2->4-x*2->8
   expanding:  1-x*2->2-x*2->4-x+1->5
17  states visited
[(None, 1), ('x*2', 2), ('x*2', 4), ('x+1', 5), ('x*2', 10)]
```

We find the same path, but visit noticeably fewer states. If we change the goal to 27, we find that we visit 564 states without DP and 119, with. If the goal is 1027, then we visit 12710 states without DP and 1150 with DP, which is getting to be a very big difference.

To experiment with depth-first search, we can make a version of the problem where the state space is limited to the integers in some range. We do this by making a subclass of the `NumberTestSM`, which remembers the maximum legal value, and uses it to restrict the set of legal inputs for a state (any input that would cause the successor state to go out of bounds just results in staying at the same state, and it will be pruned.)

```
class NumberTestFiniteSM(NumberTestSM):
    def __init__(self, goal, maxVal):
        self.goal = goal
        self.maxVal = maxVal
    def getNextValues(self, state, action):
        nextState = self.nextState(state, action)
        if abs(nextState) < self.maxVal:
            return (nextState, nextState)
        else:
            return (state, state)
```

Here's what happens if we give it a range of -20 to +20 to work in:

```
>>> smSearch(NumberTestFiniteSM(10, 20), initialState = 1, depthFirst = True,
             DP = False)
   expanding:  1
   expanding:  1--x->-1
   expanding:  1--x->-1-x+1->0
   expanding:  1--x->-1-x*2->-2
   expanding:  1--x->-1-x*2->-2--x->2
   expanding:  1--x->-1-x*2->-2--x->2-x+1->3
   expanding:  1--x->-1-x*2->-2--x->2-x+1->3--x->-3
   expanding:  1--x->-1-x*2->-2--x->2-x+1->3--x->-3-x**2->9
20  states visited
[(None, 1), ('-x', -1), ('x*2', -2), ('-x', 2), ('x+1', 3), ('-x', -3), ('x**2', 9),
```

We generate a much longer path!

We can see from trying lots of different searches in this space that (a) the DP makes the search much more efficient and (b) that the difficulty of these search problems varies incredibly widely.

6.4 Computational Complexity

To finish up this segment, let's consider the computational complexity of these algorithms. As we've already seen, there can be a huge variation in the difficulty of a problem that depends on the exact structure of the graph, and is very hard to quantify in advance. We will perform *worst-case analysis*, which tries to characterize the approximate running time of the worst possible input for the algorithm.

First, we need to establish a bit of notation. Let

- b be the *branching factor* of the graph; that is, the number of successors a node can have. If we want to be truly worst-case in our analysis, this needs to be the maximum branching factor of the graph.
- d be the *maximum depth* of the graph; that is, the length of the longest path in the graph. In an infinite space, this could be infinite.
- l be the *solution depth* of the problem; that is, the length of the shortest path from the start state to the shallowest goal state.
- n be the *state space size* of the graph; that is the total number of states in the domain.

Depth-first search, in the worst case, will search the entire search tree. It has d levels, each of which has b times as many paths as the previous one. So, there are b^d paths on the d^{th} level. The algorithm might have to visit all of the paths at all of the levels, which is about b^{d+1} nodes. But the amount of storage it needs for the agenda is only $b \cdot d$.

Breadth-first search, on the other hand, only needs to search as deep as the depth of the best solution. So, it might have to visit as many as b^{l+1} nodes. The amount of storage required for the agenda can be as bad as b^l, too.

So, to be clear, consider the numeric search problem. The branching factor $b = 5$, in the worst case. So, if we have to find a sequence of 10 steps, breadth-first search could potentially require visiting as many as $5^{11} = 48828125$ nodes!

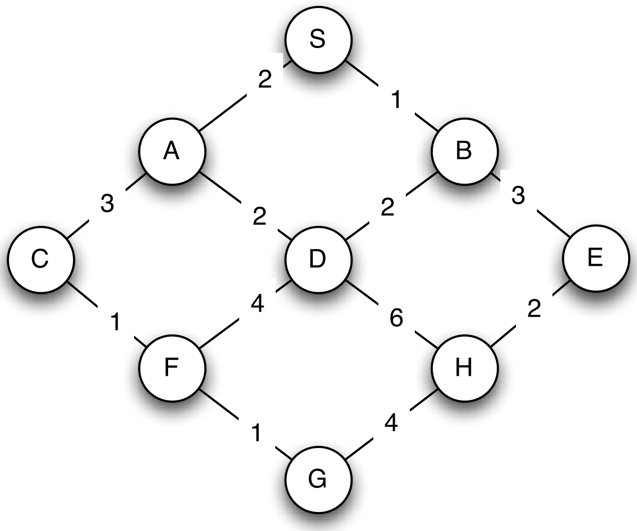
This is all pretty grim. What happens when we consider the DP version of breadth-first search? We can promise that every state in the state space is visited at most once. So, it will visit at most n states. Sometimes n is *much* smaller than b^l (for instance, in a road network). In other cases, it can be much larger (for instance, when you are solving an easy (short solution path) problem embedded in a very large space). Even so, the DP version of the search will visit fewer states, except in the very rare case in which there are never multiple paths to the same state (the graph is actually a tree). For example, in the numeric search problem, the shortest path from 1 to 91 is 9 steps long, but using DP it only requires visiting 1973 states, rather than $5^{10} = 9765625$.

6.5 Uniform-Cost Search

In many cases, the arcs in our graph will actually have different costs. In a road network, we would really like to find the shortest path in miles (or in time to traverse), and different road segments have different lengths and times. To handle a problem like this, we need to extend our representation of search problems, and add a new algorithm to our repertoire.

We will extend our notion of a successor function, so that it takes a state and an action, as before, but now it returns a pair `(newS, cost)`, which represents the resulting state, as well as the cost that is incurred in traversing that arc. To guarantee that all of our algorithms are well behaved, we will require that all costs be positive (not zero or negative).

Here is our original city map, now with distances associated with the roads between the cities.



We can describe it in a dictionary, this time associating a cost with each resulting state, as follows:

```
map1dist = {'S' : [('A', 2), ('B', 1)],
          'A' : [('S', 2), ('C', 3), ('D', 2)],
          'B' : [('S', 1), ('D', 2), ('E', 3)],
          'C' : [('A', 3), ('F', 1)],
          'D' : [('A', 2), ('B', 2), ('F', 4), ('H', 6)],
          'E' : [('B', 3), ('H', 2)],
          'F' : [('C', 1), ('D', 4), ('G', 1)],
          'H' : [('D', 6), ('E', 2), ('G', 4)],
          'G' : [('F', 1), ('H', 4)]}
```

When we studied breadth-first search, we argued that it found the shortest path, in the sense of having the fewest nodes, by seeing that it investigate all of the length 1 paths, then all of the length 2 paths, etc. This orderly enumeration of the paths guaranteed that when we first encountered a goal state, it would be via a shortest path. The idea behind *uniform cost search* is basically the same: we are going to investigate paths through the graph, in the order of the sum of the costs on their arcs. If we do this, we guarantee that the first time we extract a node with a given state from the agenda, it will be via a shortest path, and so the first time we extract a node with a goal state from the agenda, it will be an optimal solution to our problem.

Priority Queue

Just as we used a stack to manage the agenda for depth-first search and a queue to manage the agenda for bread-first search, we will need to introduce a new data structure, called a *priority queue* to manage the agenda for uniform-cost search. A priority queue is a data structure with the same basic operations as stacks and queues, with two differences:

- Items are pushed into a priority queue with a numeric score, called a *cost*.
- When it is time to pop an item, the item in the priority queue with the least *cost* is returned and removed from the priority queue.

There are many interesting ways to implement priority queues so that they are very computationally efficient. Here, we show a very simple implementation that simply walks down the entire contents of the priority queue to find the least-cost item for a pop operation. Its `data` attribute consists of a list of `(cost, item)` pairs. It calls the `argmaxIndex` procedure from our utility package, which takes a list of items and a scoring function, and returns a pair consisting of the index of the list with the highest scoring item, and the score of that item. Note that, because `argmaxIndex` finds the item with the *highest score*, and we want to extract the item with the *least cost*, our scoring function is the *negative* of the cost.

```
class PQ:
    def __init__(self):
        self.data = []
    def push(self, item, cost):
        self.data.append((cost, item))
    def pop(self):
        (index, cost) = util.argmaxIndex(self.data, lambda (c, x): -c)
        return self.data.pop(index)[1] # just return the data item
    def isEmpty(self):
        return self.data is []
```

UC Search

Now, we're ready to study the uniform-cost search algorithm itself. We will start with a simple version that doesn't do any pruning or dynamic programming, and then add those features back in later. First, we have to extend our definition of a `SearchNode`, to incorporate costs. So, when we create a new search node, we pass in an additional parameter `actionCost`, which represents the cost just for the action that moves from the parent node to the state. Then, we create an attribute `self.cost`, which encodes the cost of this entire path, from the starting state to the last state in the path. We compute it by adding the path cost of the parent to the cost of this last action, as shown by the red text below.

```
class SearchNode:
    def __init__(self, action, state, parent, actionCost):
        self.state = state
        self.action = action
        self.parent = parent
        if self.parent:
            self.cost = self.parent.cost + actionCost
        else:
            self.cost = actionCost
```

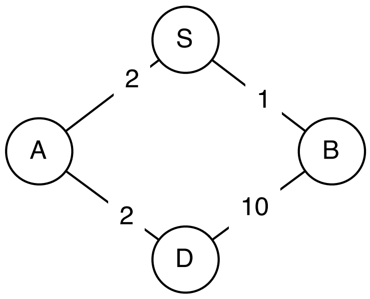
Now, here is the search algorithm. It looks a lot like our standard search algorithm, but there are two important differences:

- The agenda is a priority queue.
- Instead of testing for a goal state when we put an element *into* the agenda, as we did in breadth-first search, we test for a goal state when we take an element *out of* the agenda. This is crucial, to ensure that we actually find the shortest path to a goal state.

```
def ucSearch(initialState, goalTest, actions, successor):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    while not agenda.isEmpty():
        n = agenda.pop()
        if goalTest(n.state):
            return n.path()
        for a in actions:
            (newS, cost) = successor(n.state, a)
            if not n.inPath(newS):
                newN = SearchNode(a, newS, n, cost)
                agenda.push(newN, newN.cost)
    return None
```

Example

Consider the following simple graph:



Let's simulate the uniform-cost search algorithm, and see what happens when we try to start from `S` and go to `D`:

- The agenda is initialized to contain the starting node. The agenda is shown as a list of cost, node pairs.

```
agenda:  PQ([(0, S)])
```

- The least-cost node, `S`, is extracted and expanded, adding two new nodes to the agenda. The notation `S-0->A` means that the path starts in state `S`, takes action `0`, and goes to state `A`.

```
    0 :   expanding:  S
agenda:  PQ([(2, S-0->A), (1, S-1->B)])
```

- The least-cost node, `S-1->B`, is extracted, and expanded, adding one new node to the agenda. Note, that, at this point, we have discovered a path to the goal: `S-1->B-1->D` is a path to the goal, with cost 11. But we cannot be sure that it is the shortest path to the goal, so we simply put it into the agenda, and wait to see if it gets extracted before any other path to a goal state.

```
    1 :   expanding:  S-1->B
agenda:  PQ([(2, S-0->A), (11, S-1->B-1->D)])
```

- The least-cost node, `S-0->A` is extracted, and expanded, adding one new node to the agenda. At this point, we have two different paths to the goal in the agenda.

```
    2 :   expanding:  S-0->A
agenda:  PQ([(11, S-1->B-1->D), (4, S-0->A-1->D)])
```

- Finally, the least-cost node, `S-0->A-1->D` is extracted. It is a path to the goal, so it is returned as the solution.

```
5 states visited; Solution cost: 4
[(None, 'S'), (0, 'A'), (1, 'D')]
```

Dynamic programming

Now, we just need to add dynamic programming back in, but we have to do it slightly differently. We promise that, once we have *expanded* a node, that is, taken it out of the agenda, then we have found the shortest path to that state, and we need not consider any further paths that go through that state. So, instead of remembering which nodes we have visited (put onto the agenda) we will remember nodes we have *expanded* (gotten out of the agenda), and never visit or expand a node that has already been expanded. In the code below, the first test ensures that we don't expand a node that goes to a state that we have already found the shortest path to, and the second test ensures that we don't put any additional paths to such a state into the agenda.

```
def ucSearch(initialState, goalTest, actions, successor):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    expanded = { }
    while not agenda.isEmpty():
        n = agenda.pop()
        if not expanded.has_key(n.state):
            expanded[n.state] = True
            if goalTest(n.state):
                return n.path()
            for a in actions:
                (newS, cost) = successor(n.state, a)
                if not expanded.has_key(newS):
                    newN = SearchNode(a, newS, n, cost)
                    agenda.push(newN, newN.cost)
    return None
```

Here is the result of running this version of uniform cost search on our bigger city graph with distances:

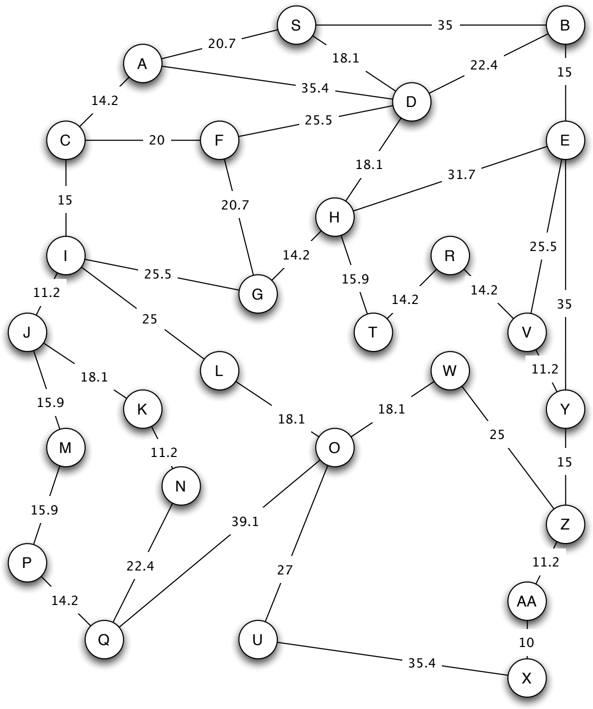
```
mapDistTest(map1dist,'S', 'G')
agenda:  PQ([(0, S)])
    0 :   expanding:  S
agenda:  PQ([(2, S-0->A), (1, S-1->B)])
    1 :   expanding:  S-1->B
agenda:  PQ([(2, S-0->A), (3, S-1->B-1->D), (4, S-1->B-2->E)])
    2 :   expanding:  S-0->A
agenda:  PQ([(3, S-1->B-1->D), (4, S-1->B-2->E), (5, S-0->A-1->C), (4, S-0
->A-2->D)])
    3 :   expanding:  S-1->B-1->D
agenda:  PQ([(4, S-1->B-2->E), (5, S-0->A-1->C), (4, S-0->A-2->D), (7, S-1
->B-1->D-2->F), (9, S-1->B-1->D-3->H)])
    4 :   expanding:  S-1->B-2->E
agenda:  PQ([(5, S-0->A-1->C), (4, S-0->A-2->D), (7, S-1->B-1->D-2->F),
(9, S-1->B-1->D-3->H), (6, S-1->B-2->E-1->H)])
agenda:  PQ([(5, S-0->A-1->C), (7, S-1->B-1->D-2->F), (9, S-1->B-1->D-3->H),
(6, S-1->B-2->E-1->H)])
    5 :   expanding:  S-0->A-1->C
agenda:  PQ([(7, S-1->B-1->D-2->F), (9, S-1->B-1->D-3->H), (6, S-1->B-2->
E-1->H), (6, S-0->A-1->C-1->F)])
    6 :   expanding:  S-1->B-2->E-1->H
agenda:  PQ([(7, S-1->B-1->D-2->F), (9, S-1->B-1->D-3->H), (6, S-0->A-1->
C-1->F), (10, S-1->B-2->E-1->H-2->G)])
    6 :   expanding:  S-0->A-1->C-1->F
agenda:  PQ([(7, S-1->B-1->D-2->F), (9, S-1->B-1->D-3->H), (10, S-1->B-2->
E-1->H-2->G), (7, S-0->A-1->C-1->F-2->G)])
agenda:  PQ([(9, S-1->B-1->D-3->H), (10, S-1->B-2->E-1->H-2->G), (7, S-0->A-1
->C-1->F-2->G)])
13 states visited; Solution cost: 7
[(None, 'S'), (0, 'A'), (1, 'C'), (1, 'F'), (2, 'G')]
```

6.5.1 Connection to State Machines

When we use a state machine to specify a domain for a cost-based search, we only need to make a small change: the `getNextValues` method of a state machine can still serve as the `successor` function in a search (the inputs to the machine are the actions). We usually think of `getNextValues` as returning the next state and the output: now, we will modify that interpretation slightly, and think of it as returning the next state and the incremental cost of taking the action that transitions to that next state. This has the form that the `ucSearch.search` procedure expects of a successor function, so we don't need to change anything about the `smSearch` procedure we have already defined.

6.6 Search with Heuristics

Ultimately, we'd like to be able to solve huge state-space search problems, such as those solved by a GPS that can plan long routes through a complex road network. We'll have to add something to uniform-cost search to solve such problems efficiently. Let's consider the city below, where the actual distances between the intersections are shown on the arcs:



If we use uniform cost search to find a path from G to X, we expand states in the following order (the number at the beginning of each line is the length of the path from G to the state at the end of the path:

```
>>> bigTest('G', 'X')
    0 :   expanding:  G
    14.2 :   expanding:  G-2->H
    20.7 :   expanding:  G-1->F
    25.5 :   expanding:  G-0->I
    30.1 :   expanding:  G-2->H-2->T
    32.3 :   expanding:  G-2->H-0->D
    36.7 :   expanding:  G-0->I-3->J
    40.5 :   expanding:  G-0->I-0->C
    44.3 :   expanding:  G-2->H-2->T-1->R
    45.9 :   expanding:  G-2->H-1->E
    50.4 :   expanding:  G-2->H-0->D-0->S
    50.5 :   expanding:  G-0->I-2->L
    52.6 :   expanding:  G-0->I-3->J-2->M
    54.7 :   expanding:  G-2->H-0->D-2->B
    54.7 :   expanding:  G-0->I-0->C-0->A
    54.8 :   expanding:  G-0->I-3->J-1->K
    58.5 :   expanding:  G-2->H-2->T-1->R-1->V
    66.0 :   expanding:  G-0->I-3->J-1->K-1->N
    68.5 :   expanding:  G-0->I-3->J-2->M-1->P
    68.6 :   expanding:  G-0->I-2->L-1->O
    69.7 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y
    82.7 :   expanding:  G-0->I-3->J-2->M-1->P-1->Q
    84.7 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y-2->Z
    86.7 :   expanding:  G-0->I-2->L-1->O-1->W
    95.6 :   expanding:  G-0->I-2->L-1->O-2->U
    95.9 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y-2->Z-2->AA
39 nodes visited; 27 states expanded; solution cost: 105.9
[(None, 'G'), (2, 'H'), (2, 'T'), (1, 'R'), (1, 'V'), (2, 'Y'), (2, 'Z'),
(2, 'AA'), (1, 'X')]
```

This search process works its way out, radially, from `G`, expanding nodes in contours of increasing path length. That means that, by the time the search expands node `X`, it has expanded every single node. This seems kind of silly: if you were looking for a good route from `G` to `X`, it's unlikely that states like `S` and `B` would ever come into consideration.

Heuristics

What is it about state `B` that makes it seem so irrelevant? Clearly, it's far away from where we want to go. We can incorporate this idea into our search algorithm using something called a *heuristic function*. A heuristic function takes a state as an argument and returns a numeric estimate of the total cost that it will take to reach the goal from there. We can modify our search algorithm to be biased toward states that are closer to the goal, in the sense that the heuristic function has a smaller value on them.

In a path-planning domain, such as our example, a reasonable heuristic is the actual Euclidean distance between the current state and the goal state; this makes sense because the states in this domain are actual locations on a map.

A*

If we modify the uniform-cost search algorithm to take advantage of a heuristic function, we get an algorithm called A^* (pronounced 'a star'). It is given below, with the differences highlighted in red. The *only* difference is that, when we insert a node into the priority queue, we do so with a cost that is `newN.cost + heuristic(newS)`. That is, it is the sum of the actual cost of the path from the start state to the current state, and the estimated cost to go from the current state to the goal.

```
def ucSearch(initialState, goalTest, actions, successor, heuristic):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
```

```
agenda = PQ()
agenda.push(startNode, 0)
expanded =
while not agenda.isEmpty():
    n = agenda.pop()
    if not expanded.has_key(n.state):
        expanded[n.state] = True
        if goalTest(n.state):
            return n.path()
        for a in actions:
            (newS, cost) = successor(n.state, a)
            if not expanded.has_key(newS):
                newN = SearchNode(a, newS, n, cost)
                agenda.push(newN, newN.cost + heuristic(newS))
return None
```

Example

Now, we can try to search in the big map for a path from `G` to `X`, using, as our heuristic function, the distance between the state of interest and `X`. Here is a trace of what happens (with the numbers rounded to increase readability):

- We get the start node out of the agenda, and add its children. Note that the costs are the actual path cost *plus* the heuristic estimate.

```
    0 :   expanding:  G
agenda:  PQ([(107, G-0->I), (101, G-1->F), (79, G-2->H)])
```

- The least cost path is `G-2->H`, so we extract it, and add its successors.

```
    14.2 :   expanding:  G-2->H
agenda:  PQ([(107, G-0->I), (101, G-1->F), (109, G-2->H-0->D), (116,
G-2->H-1->E), (79, G-2->H-2->T)])
```

- Now, we can see the heuristic function really having an effect. The path `G-2->H-2->T` has length 30.1, and the path `G-1-F` has length 20.7. But when we add in the heuristic cost estimates, the path to `T` has a lower cost, because it seems to be going in the right direction. Thus, we select `G-2->H-2->T` to expand next:

```
    30.1 :   expanding:  G-2->H-2->T
agenda:  PQ([(107, G-0->I), (101, G-1->F), (109, G-2->H-0->D), (116,
G-2->H-1->E), (100, G-2->H-2->T-1->R)])
```

- Now the path `G-2->H-2->T-1->R` looks best, so we expand it.

```
    44.3 :   expanding:  G-2->H-2->T-1->R
agenda:  PQ([(107, G-0->I), (101, G-1->F), (109, G-2->H-0->D), (116,
G-2->H-1->E), (103.5, G-2->H-2->T-1->R-1->V)])
```

- Here, something interesting happens. The node with the least estimated cost is `G-1->F`. It's going in the wrong direction, but if we were to be able to fly straight from `F` to `X`, then that would be a good way to go. So, we expand it:

```
    20.7 :   expanding:  G-1->F
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (103.5,
G-2->H-2->T-1->R-1->V), (123, G-1->F-0->D), (133, G-1->F-1->C)])
```

- Continuing now, basically straight to the goal, we have:

```
    58.5 :   expanding:  G-2->H-2->T-1->R-1->V
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (123,
G-1->F-0->D), (133, G-1->F-1->C), (154, G-2->H-2->T-1->R-1->V-1->E),
 (105, G-2->H-2->T-1->R-1->V-2->Y)])
    69.7 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (123,
G-1->F-0->D), (133, G-1->F-1->C), (154, G-2->H-2->T-1->R-1->V-1->E),
(175, G-2->H-2->T-1->R-1->V-2->Y-0->E), (105, G-2->H-2->T-1->R-1->
V-2->Y-2->Z)])
    84.7 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y-2->Z
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (123,
G-1->F-0->D), (133, G-1->F-1->C), (154, G-2->H-2->T-1->R-1->V-1->E),
(175, G-2->H-2->T-1->R-1->V-2->Y-0->E), (151, G-2->H-2->T-1->R-1->
V-2->Y-2->Z-1->W), (106, G-2->H-2->T-1->R-1->V-2->Y-2->Z-2->AA)])
    95.9 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y-2->Z-2->AA
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (123,
G-1->F-0->D), (133, G-1->F-1->C), (154, G-2->H-2->T-1->R-1->V-1->E),
(175, G-2->H-2->T-1->R-1->V-2->Y-0->E), (151, G-2->H-2->T-1->R-1->
V-2->Y-2->Z-1->W), (106, G-2->H-2->T-1->R-1->V-2->Y-2->Z-2->AA-1->X)])
18 nodes visited; 10 states expanded; solution cost: 105.9
[(None, 'G'), (2, 'H'), (2, 'T'), (1, 'R'), (1, 'V'), (2, 'Y'), (2, 'Z'), (2, 'AA')
```

Using A* has roughly halved the number of nodes visited and expanded. In some problems it can result in an enormous savings, but, as we'll see in the next section, it depends on the heuristic we use.

Good and bad heuristics

In order to think about what makes a heuristic good or bad, let's imagine what the perfect heuristic would be. If we were to magically know the distance, via the shortest path in the graph, from each node to the goal, then we could use that as a heuristic. It would lead us directly from start to goal, without expanding any extra nodes. But, of course, that's silly, because it would be at least as hard to compute the heuristic function as it would be to solve the original search problem.

So, we would like our heuristic function to give an estimate that is as close as possible to the true shortest-path-length from the state to the goal, but also to be relatively efficient to compute.

An important additional question is: if we use a heuristic function, are we still guaranteed to find the shortest path through our state space? The answer is: yes, if the heuristic function is *admissible*. A heuristic function is admissible if it is guaranteed to be an *underestimate* of the actual cost of the optimal path to the goal. To see why this is important, consider a state `s` from which the goal can actually be reached in 10 steps, but for which the heuristic function gives a value of 100. Any path to that state will be put into the agenda with a total cost of 90 more than the true cost. That means that if a path is found that is as much as 89 units more expensive that the optimal path, it will be accepted and returned as a result of the search.

It is important to see that if our heuristic function always returns value 0, it is admissible. And, in fact, with that heuristic, the A^* algorithm reduces to uniform cost search.

In the example of navigating through a city, we used the Euclidean distance between cities, which, if distance is our cost, is clearly admissible; there's no shorter path between any two points.

Exercise 6.1
Would the so-called 'Manhattan distance', which is the sum of the absolute differences of the x and y coordinates be an admissible heuristic in the city navigation problem, in general?

Exercise 6.2
If we were trying to minimize travel time on a road network (and so the estimated time to travel each road segment was the cost), what would be an appropriate heuristic function?