

How to Think Like a Computer Scientist:  
Learning to Program with Boo

By J. Bryan Kelly  
Revision 2

Edited By: Rolly Noel

© 2012 Lulu Author. All rights reserved.  
ISBN 978-1-105-93576-3

Original "How to Think List a Computer Scientist: Learning to Program with Python" By ©2002, Green Tea Press, Authors: Allen Downey, Jeffrey Elkner, and Chris Meyers

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being "Foreword", "Preface", and "Contributor List," with no Front-Cover Texts, and with no Back Cover Texts. A copy of the license is included in the appendix entitled "GNU Free Documentation License." The GNU Free Documentation License is available from [www.gnu.org](http://www.gnu.org) or by writing to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.



## Table of Contents

Preface.....	4
Chapter 1: The Way of the Program.....	7
Chapter 2: Variables, Expressions, and Statements.....	19
Chapter 3: Functions.....	21
Chapter 4: Conditionals and Recursions.....	38
Chapter 5: Fruitful functions.....	51
Chapter 6: Iteration.....	65
Chapter 7: Strings.....	81
Chapter 8: Lists.....	93
9 .....	108
Chapter 10: Hashes.....	109
Chapter 11: Files and Exceptions.....	119
Chapter 12: Classes and objects.....	131
Chapter 13: Classes and functions.....	145
Chapter 14: Classes and methods.....	154
Chapter 15.0: Sets of objects.....	166
Chapter 16.0: Inheritance.....	176
Chapter 17.0: Linked lists.....	196
Chapter 18.0: Stacks.....	210
Chapter 19.0: Queues.....	219
Chapter 20.0: Trees.....	230
Appendix A: Debugging.....	249
Appendix B: GNU Free Documentation License.....	260
GNU Free Documentation.....	261

## **Preface**

This book is intended for the novice programmer as an introduction into the basics of computer programming.

Boo's roots lie in another programming language named Python. Part of Python's success is the wealth of documentation available for the beginning programmer. One particularly influential work is "How to Think Like a Computer Scientist: Learning to Program with Python" which introduces the curious to the world of computer programming. This is an adaptation of that book for the Boo programming language.

The Boo language is well suited to the entry level programmer to get started programming in the Microsoft .Net environment. My goal is for anyone interested in programming to find this text a useful tool for discovering software development.

### ***How and why I Came to use Boo.***

An MSCE named "Mike" had given me some wisdom that I did not follow early in my career. He pointed out that there is lots of great technology out there. The smart thing to do is to narrow down your interest to those technologies directly attributable to your career. The nuances of the technologies create learning difficulty and it is very difficult to advance studying one technology part-time during your evenings and learning another during the workday.

For example, If you spent at least 40 hours per week working for a company that was using the Java platform and you were interested in the Microsoft .Net platform. Then you have two options to being successful. Option 1, is give up your interest in .Net and spend additional time with Java or number two, find

another job where the company utilizes .Net. I have a friend in the Bay area and I never told him this story, but he knew from the day he started working that we was interested in developing in Open Source environments. Pretty soon he was moving from job to job (and sometimes between jobs). At present, he works for a local university developing open source software and is extremely happy with his career decisions.

## ***Introducing Programming with Boo.***

The Boo language greatly simplifies programming examples and makes important programming ideas easier to teach. The first example from the text illustrates this point. It is the traditional "hello, world" program, which in a language known as C++ looks like this:

```
#include <iostream>
void main()
{
    std::cout << "Hello, world.\n";
}
```

In the Boo world it becomes:

```
print "hello world"
```

Even though this is a trivial example, the advantages of Boo stand out. Boo is inherently more readable by the lay person. Secondly, Boo is free of the numerous special characters used to write the C++ version of the code. The lack of special characters in the Boo example make the code simpler to write and easier to learn.

Special thanks to the following list of individuals who made this book possible.

## ***Contributor List***

[dev@boo.codeshaus.org](mailto:dev@boo.codeshaus.org) – Thanks for the many near instantaneous answers to my frequent questions on the list. Special thanks to “Bet's On” for being the first to respond on so many occasions.

Thanks to Allen Downey, Jeffrey Elkner, and Chris Meyers for the original Python text that served as the basis and much of the copy for this document.

Thanks to Joan for her steadfast support.

## Chapter 1: The Way of the Program.

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating trade-offs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is problem solving. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The way of the program." On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

### ***1.1 The Boo programming language***

Boo is an example of a high-level language; other high-level languages you might have heard of are C#, VB.Net, Python, and Java.

As you might infer from the name "high-level language," there are also low-level languages, sometimes referred to as machine languages or assembly languages.

Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run. This extra



processing takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are portable, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

Due to these advantages, most programs today are written in high-level languages. Low-level languages are used only for a few specialized applications. Two kinds of programs process high-level languages into low-level languages: interpreters and compilers. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.

SOURCE CODE --> INTERPRETER → EXECUTION →  
OUTPUT

A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the source code, and the translated program is called the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation.

SOURCE CODE --> COMPILER → OBJECT PROGRAM →  
EXECUTE PROGRAM → OUTPUT

There are four ways to use boo:

**booish** is the interpreter: With booish, you type Boo programs

and the interpreter prints the result:

Welcome to Booish, an interactive interpreter for the Boo programming language.  
Running boo 0.9.4.9 in CLR 2.0.50727.5448.

Enter boo code in the prompt below (or type /help).

```
>>> print 1 + 1
2
>>>
```

You can follow along the example above and the rest of the examples in this book by installing “boo” on your computer. To install boo visit the Boo home page (<http://boo.codehaus.org>) and follow the links to download Boo (its free). In the docs folder (in the directory Boo is installed in) you will find the BooManifesto (a very good read). Double-click on booish (in the bin directory).

The first lines are messages from the interpreter. The `>>>` line is the prompt the interpreter uses to indicate that it is ready. We typed `print 1 + 1`, and the interpreter replied 2.

Alternatively, you can write a program in a file and use **booi.exe** to execute the contents of the file. Such a file is called a script. For example, we used a text editor to create a file named `chapter1.1.boo` with the following contents:

```
print 1 + 1
```

By convention, files that contain boo programs have names that end with `.boo`. To execute the program, we have to tell the interpreter the name of the script.

To start, open a command window (or shell) and change your directory to your `Boo\bin` folder. Then type:

## *Learning to Program with Boo*

```
> booi c:\source\chapter1.1.boo  
2
```

You can use the up arrow and Enter keys in the command window to re-run booi on your program after having changed the source code and re-saved it in the text editor. In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.

The third way is to use **booc** (from a command window as above) which creates a compiled .exe file based on your code.

```
booc c:\source\chapter1.1.boo
```

This will cause the application chapter1.1.exe to be saved by default in the Boo\bin directory. This file can be executed without any further need to compile it.

A fourth way is to download an Integrated Development Environment like SharpDevelop or MonoDevelop. These programs provide advanced features for writing software and support Boo.

We will be using a hybrid of the first and second method in this text. - Write your code in a text editor (like Notepad). Save your source code in files that end in “.boo”. Then execute your code by passing the file in as a /l argument to booish.exe. Let's try an example!

- Create a file named "learn.boo". Save it to a folder such as [c:\source](#)
- Open a command window and change your directory to your Boo\bin folder
- Then type: booish
- Write your boo code inside learn.boo (you can copy & paste it from this ebook). For example: print 1 + 1
- Save it

- Switch back to the command window
- Type `interpreter.Reset()` if you need to clear booish of previous symbol definitions
- Type `/load` and the file name:  
`>>> /load c:\source\learn.boo`  
`>>>`
- Make any changes in your text editor & save the file, type `/l learn.boo` again to execute your changes. You can abbreviate the `/load` command as `/l` and Boo will still understand you are loading a file.
- Save the code under another file name if you want to keep it.
- If your keyboard driver supports it, it is handy to define a macro for a special key that contains the `interpreter.Reset()` and load commands.
- To have a console application's window stay around so you can view its output, end the Boo program with the lines:  
`print "Hit any key to exit"`  
`System.Console.ReadKey(true)`<sup>1</sup>

## 1.2 What is a program

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

---

<sup>1</sup> The examples in this book were tested with Boo version 0.9.4.9 and SharpDevelop 4.2's version 0.9.4.3504 of Boo.

input: Get data from the keyboard, a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

conditional execution: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look more or less like these. Thus, we can describe programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions. That may be a little vague, but we will come back to this topic later when we talk about algorithms.

### ***1.3 What is debugging?***

Programming is a complex process, and because it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called bugs and the process of tracking them down and correcting them is called debugging. Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

### 1.3.1 Syntax errors

Boo can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. Syntax refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Boo is not so forgiving. If there is a single syntax error anywhere in your program, Boo will print an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and correct them faster.

### 1.3.2 Runtime errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

### 1.3.3 Semantic errors

The third type of error is the semantic error. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. The problem is that the program you wrote is not the program you wanted to write. The meaning of the

program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

### **1.3.4 Experimental debugging**

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see. Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does something and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenweld, one of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux." (*The Linux Users' Guide Beta*

*Version 1)* Later chapters will make more suggestions about debugging and other programming practices.

## **1.4 Formal and natural languages**

Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for special applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly: Programming languages are formal languages that have been designed to express computations. Formal languages tend to have strict rules about syntax. For example,  $3+3 = 6$  is a syntactically correct mathematical statement, but  $3=+6\$$  is not.  $\text{H}_2\text{O}$  is a syntactically correct chemical name, but  $2\text{Zz}$  is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with  $3=+6\$$  is that  $\$$  is not a legal token in mathematics (at least as far as we know). Similarly,  $2\text{Zz}$  is not legal because there is no element with the abbreviation  $\text{Zz}$ .

The second type of syntax error pertains to the structure of a statement that is, the way the tokens are arranged. The statement  $3=+6\$$  is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before. As an exercise, create what appears to be a well-structured English sentence with unrecognizable tokens in it. Then write another



sentence with all valid tokens but with invalid structure.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called parsing. For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a shoe is and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common (tokens, structure, syntax, and semantics), there are many differences:

**ambiguity:** Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

**redundancy:** In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

**literalness:** Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

**Poetry:** Words are used for their sounds as well as for their meaning, and the whole poem together creates an emotional response. Ambiguity is not only common but often deliberate.

**Prose:** The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

**Programs:** The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages).

First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

## ***1.5 The first program***

Traditionally, the first program written in a new language is called "Hello, World!" because all it does is display the words, "Hello, World!" In boo, it looks like this:

```
print "Hello, World!"
```

This is an example of a print statement, which doesn't actually print anything on paper. It displays a value on the screen. In this

case, the result is the words *Hello, World!* The quotation marks in the program mark the beginning and end of the value; they don't appear in the result.

Some people judge the quality of a programming language by the simplicity of the "Hello, World!" program. By this standard, boo does about as well as possible.

## **1.6 Glossary**

problem solving: The process of formulating a problem, finding a solution, and expressing the solution.

high-level language: A programming language like Boo that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to execute; also called "machine language" or "assembly language."

portability: A property of a program that can run on more than one kind of computer.

interpret: To execute a program in a high-level language by translating it one line at a time.

compile: To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

source code: A program in a high-level language before being compiled.

object code: The output of the compiler after it translates the program.

**executable:** Another name for object code that is ready to be executed.

**script:** A program stored in a file (usually one that will be interpreted).

**program:** A set of instructions that specifies a computation.

**algorithm:** A general process for solving a category of problems.

**bug:** An error in a program.

**debugging:** The process of finding and removing any of the three kinds of programming errors.

**syntax:** The structure of a program.

**syntax error:** An error in a program that makes it impossible to parse (and therefore impossible to interpret).

**runtime error:** An error that does not occur until the program has started to execute but that prevents the program from continuing.

**exception:** Another name for a runtime error.

**semantic error:** An error in a program that makes it do something other than what the programmer intended.

**semantics:** The meaning of a program.

**natural language:** Any one of the languages that people speak that evolved naturally.

**formal language:** Any one of the languages that people have designed for special purposes, such as representing mathematical ideas or computer programs; all programming languages are

formal languages.

token: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

parse: To examine a program and analyze the syntactic structure.

print statement: An instruction that causes the Boo interpreter to display a value on the screen.

## **Chapter 2: Variables, Expressions, and Statements**

### ***2.1 Values and types***

A value is one of the fundamental things -- like a letter or a number -- that a program manipulates. The values we have seen so far are 2 (the result when we added  $1 + 1$ ), and "Hello, World!". These values belong to different types: 2 is an integer, and "Hello, World!" is a string, so-called because it contains a "string" of characters. You (and the interpreter) can identify strings because they are enclosed in quotation marks. The print statement also works for integers.

```
>>> print 42  
4
```

### ***2.2 Variables***

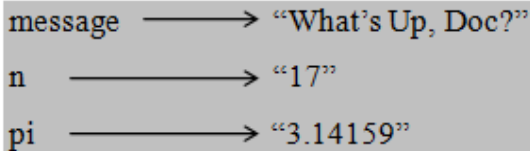
One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value. The assignment statement creates new variables and gives them values:

---

<sup>2</sup> Beyond this point in the code samples, booish's >>> prompt will not be shown, and any output will be shown as a comment on the line that produces it. As a result, code can more easily be copied & pasted to try it out.

```
message = "What's up, Doc?"
n = 17
pi = 3.14159
```

The code examples makes three assignments. The first assigns the string "What's up, Doc?" to a new variable named *message*. The second gives the integer 17 to *n*, and the third gives the floating-point number 3.14159 to *pi*. A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a state diagram because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the assignment statements:



```
message  —————> "What's Up, Doc?"
n        —————> "17"
pi       —————> "3.14159"
```

The `print` statement also works with variables.

```
print message #outputs: What's up, Doc?
print n       #outputs: 17
print pi      #outputs: 3.14159
```

<p>The “#” symbol can be used to add a comment to your source code. The comment is human readable but will be ignored by the computer.</p>
--

<p>In each case the result is the value of the variable. Variables also have types; again, we can ask the interpreter what they are.</p>
--

```
print message.GetType()#out's: System.String
```

## Chapter 3: Functions

### 3.1 Function calls

You have already seen one example of a function call:

```
print("32")      #outputs: 32
```

The name of the function is `print`, and it displays the string of characters passed to it. The value or variable, which is called the argument of the function, has to be enclosed in parentheses<sup>3</sup>. It is common to say that a function "takes" an argument and "returns" a result. The result is called the return value. Instead of printing the return value, we could assign it to a variable:

```
spacedOut = join("tootight")
print spacedOut    #outputs: t o o t i g h t
```

As another example, the `BooVersion` function takes no variables and returns the version of Boo you are currently using.

```
print BooVersion    #outputs: 0.9.4.9
```

### 3.2 Types

Types define what a values or a variable is. Looking back at Section 2, the first thing we did was "print 4". 4 to a human is a number. Computers are more specific than people. A number can be more one of several types. The number 4 is an 'int'. 'int' is short for *integer*. Another number type that we will commonly use is called 'double'. We will use `double` when we want to declare a variable to store a number that contains values to the right of the decimal.

---

<sup>3</sup> Boo doesn't mind you forgoing the parentheses in the case of the `print` function.

Boo is a statically typed language. This means every time a variable is created, its type should be declared. This helps the compiler create executable code that will run faster and eliminates some types of coding errors. Because Boo seeks to be helpful, it provides Type Inference to ease the burden created by the static typing requirement. Type Inference means that if you do not define a type, Boo will select a default type based on the assignment to the variable.

Duck typing is available by default in booish but it must be enabled if using booish or booc. In the early examples in chapters 1-5 we will declare types. If you are using booish you can choose to include them or not. Beginning in chapter 6 we must declare types as the examples become more

“If it walks like a Duck,  
and quacks like a Duck, it  
must be a Duck.”

complex. Boo provides a wide variety of built-in types. We will introduce the following types in the subsequent chapters of this book:

**int:** Use `int` to declare an integer. An integer is whole number that may be positive, negative, or zero.

**bool:** Use `bool` to declare a boolean. A boolean is a flag that may be set to 'true' or 'false'. It is good to use boolean in answering 'yes' or 'no' questions.

**single:** Allows a number to have a decimal.

**double:** An alternative way to allow a number to have a decimal, but allows for larger numbers at the expense of memory.

**string:** A string is a series of characters.

**char:** A single character.



Some examples<sup>4</sup>:

```
x as int
x = 10
print x           #outputs: 10
x = 10.1
print x           #outputs: 10
y as double
y = 10
print y           #outputs: 10
y = 10.1
print y           #outputs: 10.1
print x + y       #outputs: 20.1
```

In the first example, *x* is defined as an integer. When we assigned a value with a decimal, only the integer portion was remembered by Boo. In the second example we defined *y* as a double. This time the entire value is recalled when we print. In the final example when we print *x + y*. Notice (if keying these examples into booish) that when we first declared *x* and *y*, they were assigned values of 0. Then we assigned a value to them by adding an expression with an '=' sign. This can be neatly done in a single step.

```
z as double = 20.2
print z       #outputs: 20.2
```

In later chapters we will learn how to build our own user-defined types called structures. Occasionally you will want to know what type a variable is. An easy way to do this is to append “.GetType()” to the variable name.<sup>5</sup>

---

<sup>4</sup> When typing in code to booish, any statement defining a variable (e.g. “*x* as int” in the following example), or assigning a value to it (“*x* = 10”), outputs the value to the console window. This does not occur when using booish's /load command, nor when running boo code in SharpDevelop. This extra output will not be shown in the examples in this book.

<sup>5</sup> For example: “*z*.GetType()” if keying in to booish, “print *z*.GetType()”

### 3.3 Changing Types

In the example, we declared  $x$  as an integer with a value of 10 and  $y$  &  $z$  as doubles.

```
x = 10
y = 10.1
z = 20.2
```

Then lets try the following:

```
print z = x + y      #outputs: 20.1
print z = x / y      #outputs:0.990099009999
print z.GetType()    #outputs: System.Double
```

Boo allows mathematical operations on different number types, and automatically converts the result to the needed type. In this case a double. While Boo can convert number types, it cannot automatically convert strings. The following will produce an error.

```
zz as string = '4'
z = zz + x + y
ERROR: Cannot convert 'string' to 'double'.
```

To solve this problem you need to parse the string and convert it to a number.

```
print z = System.Convert.ToDouble(zz) + x+y
#the above outputs: 24.1
```

Now that we can convert between types, we have another way to deal with integer division. Suppose we want to calculate the fraction of an hour that has elapsed. The most obvious expression,

---

otherwise.

`minute / 60`, does integer arithmetic (if `minute` is defined as `int`), so the result is always 0, even at 59 minutes past the hour. One solution is to convert `minute` to a double and do the division:

```
minute = 59
print System.Convert.ToDouble(minute) / 60
#the above outputs: 0.983333333333
```

Alternatively, we can take advantage of the rules for automatic type conversion, which is called type coercion. For the mathematical operators, if either operand is a double, the other is automatically converted to a double.

```
minute = 59
minute / 60.0 #outputs: 0.983333333333
```

By making the denominator a float, we force Boo to do floating-point division.

### **3.4 Math functions**

In mathematics, you have probably seen functions like `sin` and `log`, and you have learned to evaluate expressions like `sin(pi/2)` and `log(1/x)`. First, you evaluate the expression in parentheses (the argument). For example, `pi/2` is approximately 1.571, and `1/x` is 0.1 (if `x` happens to be 10.0). Then, you evaluate the function itself, either by looking it up in a table or by performing various computations. The `sin` of 1.571 is 1, and the `log` of 0.1 is -1 (assuming that `log` indicates the logarithm base 10). This process can be applied repeatedly to evaluate more complicated expressions like `log(1/sin(pi/2))`. First, you evaluate the argument of the innermost function, then evaluate the function, and so on. Boo inherits all the mathematical features built-in to the .Net framework. The class `System.Math` contains the constant *pi*, the natural log *e*, and over 20 common mathematical functions

such as `sqrt`, `sin`, and `min`. Functions that are attached to a class are known as methods. We will talk more about what classes and methods are in a later chapter. For now, let's learn to take advantage of `System.Math`.

```
print System.Math.PI
#the above outputs: 3.1459265358979
```

To call one of the methods, we have to specify the name of the class and the name of the method, separated by a dot, also known as a period. This format is called dot notation.

To save typing `System.Math`, before calling any of the fields or methods in `System.Math`, use the `import` command.

```
import System.Math
print PI          #outputs: 3.1459265358979
print decibel = Log10 (17.0)
#the above outputs: 1.230448921370827
angle = 1.5
print height = Sin(angle)
#the above outputs: 0.997494986604054
```

The above sets the variable `decibel` to the logarithm of 17, base 10 and prints it out. There is also a function called `log` that takes logarithm base *e*. Then we find and print out the sin of the value of the variable `angle`. `Sin` and the other trigonometric functions (`Cos`, `Tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by  $2\pi$ . For example, to find the sine of 45 degrees, first calculate the angle in radians and then take the sin:

```
import System.Math
degrees = 45
print angle = degrees * 2 * PI / 360.0
#the above outputs: 0.785398163397448
print Sin(angle) #outputs: 0.707106781186547
```

If you know your geometry, you can check the previous result by comparing it to the square root of two divided by two:

```
import System.Math
print Sqrt(2) / 2.0
#the above outputs: 0.707106781186548
```

### **3.5 Composition**

Just as with mathematical functions, Boo functions can be composed, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function:

```
x = Cos(angle + PI/2)
```

This statement takes the value of  $\pi$ , divides it by 2, and adds the result to the value of angle. The sum is then passed as an argument to the cos function. You can also take the result of one function and pass it as an argument to another:

```
x = Exp(Log(10.0))
```

This statement finds the log base e of 10 and then raises e to that power. The result gets assigned to x.

### **3.6 Adding new functions**

So far, we have only been using the functions that come with Boo, but it is also possible to add new functions. Creating new functions to solve your particular problems is one of the most useful things about a general-purpose programming language. In the context of programming, a function is a named sequence of statements that performs a desired operation. This operation is

specified in a function definition. The functions we have been using so far have been defined for us, and these definitions have been hidden. This is a good thing, because it allows us to use the functions without worrying about the details of their definitions. The syntax for a function definition is:

```
def NAME( LIST OF PARAMETERS ) :
    STATEMENTS
```

You can make up any names you want for the functions you create, except that you can't use a name that is a Boo keyword. The list of parameters specifies what information, if any, you have to provide in order to use the new function. There can be any number of statements inside the function, but they have to be indented from the left margin by the same amount. In the examples in this book, we will use an indentation of two spaces. In a more normal programming environment, using tab characters set to expand by 4 or 5 spaces would be preferable. The first couple of functions we are going to write have no parameters, so the syntax looks like this:

```
def newLine() :
    print
```

This function is named `newLine`. The empty parentheses indicate that it has no parameters. It contains only a single statement, which outputs a newline character. (That's what happens when you use a `print` command without any arguments.) The syntax for calling the new function is the same as the syntax for built-in functions:

```
print "First Line." #outputs: First line.
newLine()           #outputs a blank line
print "Second Line." #outputs: Second Line.
```

Notice the extra space between the `newLine` function and the

`print "Second Line."` expression. What if we wanted more space between the lines? We could call the same function repeatedly:

```
print "First Line." #outputs: First Line.
newLine()           #outputs a blank line
newLine()           #outputs a blank line
newLine()           #outputs a blank line
print "Second Line." #outputs: Second Line.
```

Or we could write a new function named `threeLines` that prints three new lines:

```
def threeLines():
    newLine()
    newLine()
    newLine()

print "First Line." #outputs: First Line.
threeLines()         #outputs 3 blank lines
print "Second Line." #outputs: Second Line.
```

This function contains three statements, all of which are indented by two spaces. Since the next statement is not indented, Boo knows that it is not part of the function. You should notice a few things about this program: 1.) You can call the same procedure repeatedly. In fact, it is quite common and useful to do so. 2.) You can have one function call another function; in this case `threeLines` calls `newLine`. So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two: Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command and by using English words in place of arcane code.

### 3.7 Definitions

Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call `threeLines` three times. As an exercise, write a function called `nineLines` that uses `threeLines` to print nine blank lines.

Pulling together the code fragments from Section 3.6, the whole program looks like this:

```
def newLine():
    print

def threeLines():
    newLine()
    newLine()
    newLine()

def nineLines():
    threeLines()
    threeLines()
    threeLines()

print "First Line." #outputs: First Line.
nineLines()         #outputs 9 blank lines
print "Second Line." #outputs: Second Line.
```

This program contains three functions: `newLine`, `threeLines`, and `nineLines`. The statements inside a function do not get executed until the function is called. As you might expect, you have to create a function before you can execute it. In other words, the function has to be defined before the first time it is called. As an exercise, move the last three lines of this program to the top, so the function calls appear before the definitions. Run the program and see what error message you get.



As another exercise, start with the working version of the program and move the definition of `newLine` after the definition of `threeLines`. What happens when you run this program?

### **3.8 Flow of execution**

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the flow of execution. Execution always begins at the first statement of the program<sup>6</sup>. Statements are executed one at a time, in order from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called. Although it is not common, you can define one function inside another. In this case, the inner definition isn't executed until the outer function is called. Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off. That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function! Fortunately, Boo is adept at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates. What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution.

### **3.9 Parameters and arguments**

Some of the built-in functions you have used require arguments, the values that control how the function does its job. For example,

---

<sup>6</sup> Technically, execution starts at the first line of a program that is outside of a function definition.

if you want to find the sin of a number, you have to indicate what the number is. Thus, `Sin` takes a numeric value as an argument. Some functions take more than one argument. For example, `Pow` takes two arguments, the base and the exponent. Inside the function, the values that are passed get assigned to variables called parameters. Here is an example of a user-defined function that takes an argument:

```
def printTwice(bruce):
    print bruce, bruce
```

This function takes a single argument and assigns it to a parameter named *bruce*. The value of the parameter (at this point we have no idea what it will be) is printed twice, followed by a newline. The name *bruce* was chosen to suggest that the name you give a parameter is up to you, but in general, you want to choose something more illustrative than *bruce*. The function `printTwice` works for any type that can be printed:

```
printTwice('Spam') #outputs: Spam Spam
printTwice(5)      #outputs: 5 5
printTwice(3.14159)
#the above outputs: 3.14159 3.14159
```

In the first function call, the argument is a string. In the second, it's an integer. In the third, it's a double. The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `printTwice`:

```
printTwice('Spam'*4)
#the above outputs: SpamSpamSpamSpam
SpamSpamSpamSpam
printTwice(System.Math.Cos(System.Math.PI))
#the above outputs: -1 -1
```

As usual, the expression is evaluated before the function is run, so `printTwice` prints `SpamSpamSpamSpam`

`SpamSpamSpamSpam` instead of `'Spam'*4`

`'Spam'*4`. As an exercise, write a call to

`printTwice` that does print `'Spam'*4 'Spam'*4`. Hint:

strings can be enclosed in either single or double quotes, and the type of quote not used to enclose the string can be used inside it as part of the string. We can also use a variable as an argument:

```
michael = 'Eric, the half a bee.'
printTwice(michael)
#the above outputs: Eric, the half a bee.
Eric, the half a bee.
```

Notice something very important here. The name of the variable we pass as an argument (*michael*) has nothing to do with the name of the parameter (*bruce*). It doesn't matter what the value was called back home (in the caller); here in `printTwice`, we call everybody *bruce*.

Variables and parameters are local. When you create a local variable inside a function, it only exists inside the function, and you cannot use it outside. For example:

```
def catTwice(par1 as string, par2 as string):
    cat = par1 + par2
    printTwice(cat)
```

This function takes two arguments, concatenates them, and then prints the result twice. We can call the function with two strings:

```
chant1 = "Pie Jesu domine, "
chant2 = "Dona eis requiem."
catTwice(chant1, chant2)
#the above outputs: Pie Jesu domine, Dona
eis requiem. Pie Jesu domine, Dona eis
```

```
requiem.
```

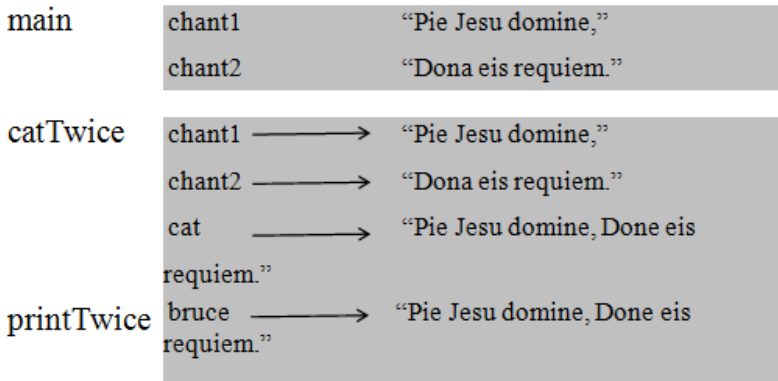
When `catTwice` terminates, the variable `cat` is destroyed. If we try to print it, we get an error:

```
print cat
ERROR: Unknown identifier: 'cat'.
```

Parameters are also local. For example, outside the function `printTwice`, there is no such thing as `bruce`. If you try to use it, Boo will complain.

### 3.10 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function to which each variable belongs. Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:



The order of the stack shows the flow of execution.

`printTwice` was called by `catTwice`, and `catTwice` was called by `main`, which is a special name for the topmost function. When you create a variable outside of any function, it belongs to `main`. Each parameter refers to the same value as its corresponding argument. So, `par1` has the same value as `chant1`, `par2` has the same value as `chant2`, and `bruce` has the same value as `cat`.

You might have noticed by now that some of the functions we are using, such as the math functions, yield results. Other functions, like `newLine`, perform an action but don't return a value. That raises some questions:

1. What happens if you call a function and you don't do anything with the result (i.e., you don't assign it to a variable or use it as part of a larger expression)?
2. What happens if you use a function without a result as part of an expression, such as `newLine() + 7`?
3. Can you write functions that yield results, or are you stuck with simple function like `newLine` and `printTwice`?

The answer to the third question is yes, and we'll do it in Chapter 5. As an exercise, answer the other two questions by trying them out. When you have a question about what is legal or illegal in Boo, a good way to find out is to ask the interpreter.

### **3.11 Glossary**

*function call*: A statement that executes a function. It consists of the name of the function followed by a list of arguments enclosed in parentheses.

*argument*: A value provided to a function when the function is

called. This value is assigned to the corresponding parameter in the function.

*return value*: The result of a function. If a function call is used in an expression, the return value is provided to the expression.

*type conversion*: An explicit statement that takes a value of one type and computes a corresponding value of another type.

*module*: A file that contains a collection of related functions and classes.

*dot notation*: The syntax for calling a function in another module, specifying the module name followed by a dot (period) and the function name.

*function*: A named sequence of statements that performs some useful operation. Functions may or may not take parameters and may or may not produce a result.

*function definition*: A statement that creates a new function, specifying its name, parameters, and the statements it executes.

*flow of execution*: The order in which statements are executed during a program run.

*parameter*: A name used inside a function to refer to the value passed as an argument.

*local variable*: A variable defined inside a function. A local variable can only be used inside its function.

*stack diagram*: A graphical representation of a stack of functions, their variables, and the values to which they refer.

*frame*: A box in a stack diagram that represents a function call. It

*Learning to Program with Boo*

contains the local variables and parameters of the function.

## Chapter 4: Conditionals and Recursions

### ***4.1 The modulus operator***

The modulus operator works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second. In Boo, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

```
print quotient = 7 / 3    #outputs: 2
print remainder = 7 % 3  #outputs: 1
```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another -- if `x % y` is zero, then `x` is divisible by `y`.

Also, you can extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

### ***4.2 Boolean expressions***

A boolean expression is an expression that is either true or false. In Boo, an expression that is true has the value 'True', and an expression that is false has the value 'False'.

The operator `==` compares two values and produces a boolean expression:

```
print 5 == 5    #outputs: True
print 5 == 6    #outputs: False
```



In the first statement, the two operands are equal, so the expression evaluates to true; in the second statement, 5 is not equal to 6, so we get false. The `==` operator is one of the comparison operators; the others are:

```
x != y      # x is not equal to y
x > y       # x is greater than y
x < y       # x is less than y
x >= y      # x is greater than or equal to y
x <= y      # x is less than or equal to y
```

Although these operations are probably familiar to you, the Boo symbols are different from the mathematical symbols. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a comparison operator. Also, there is no such thing as `=<` or `=>`.

### **4.3 Logical operators**

There are three logical operators: and, or, and not. Logical operators may only be applied to boolean variables in Boo. The semantics (meaning) of these operators is similar to their meaning in English. For example, `((x > 0) and (x < 10))` is true only if `x` is greater than 0 and less than 10. `((n%2 == 0) or (n%3 == 0))` is true if either of the conditions is true, that is, if the number is divisible by 2 or 3. Finally, the not operator negates a boolean expression, so `(not(x > y))` is true if `(x > y)` is false, that is, if `x` is less than or equal to `y`. Notice that in each example the logical expression is wrapped in parentheses. This is required by the Boo parser.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Boo is not very strict. Any nonzero number is interpreted as true, while 0 is interpreted as false.

```
x = 5
print (x and 1)      #outputs: 1
y = 0
print (y and 1)      #outputs: 0
```

In general, this sort of thing is not considered good style. If you want to compare a value to zero, you should do it explicitly.

## 4.4 *Conditional execution*

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement:

```
x = 1
if x > 0:          #outputs: x is positive
    print "x is positive"
```

The boolean expression after the if statement is called the condition. If it is true, then the indented statement gets executed. If not, nothing happens. Like other compound statements, the if statement is made up of a header and a block of statements:

```
HEADER:
    FIRST STATEMENT
    ...
    LAST STATEMENT
```

The header begins on a new line and ends with a colon (:). The indented statements that follow are called a block. The first unindented statement marks the end of the block. A statement block inside a compound statement is called the body of the statement. There is no limit on the number of statements that can appear in the body of an if statement, but there has to be at least

one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

## **4.5 Alternative execution**

A second form of the `if` statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0:
    print x, "is even"
else:
    print x, "is odd"
```

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution. As an aside, if you need to check the parity (evenness or oddness) of numbers often, you might “wrap” this code in a function:

```
def printParity(x as int):
    if x%2 == 0:
        print x, "is even"
    else:
        print x, "is odd"
```

For any value of `x`, `printParity` displays an appropriate message. When you call it, you can provide any integer expression as an argument.

```
printParity(17)
```

```
printParity(y+1)
```

## 4.6 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

```
if x < y:
    print x, "is less than", y
elif x > y:
    print x, "is greater than", y
else:
    print x, "and", y, "are equal"
```

`elif` is an abbreviation of “else if.” Again, exactly one branch will be executed. There is no limit of the number of `elif` statements, but if you have an `else` statement, it must be the last branch:

```
if choice == 'A':
    functionA()
elif choice == 'B':
    functionB()
elif choice == 'C':
    functionC()
else:
    print "Invalid choice."
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes. As an exercise, wrap these examples in functions called `compare(x, y)` and `dispatch(choice)`.

## **4.7 Nested conditionals**

One conditional can also be nested within another. We could have written the trichotomy example as follows:

```
if x == y:
    print x, "and", y, "are equal"
else:
    if x < y:
        print x, "is less than", y
    else:
        print x, "is greater than", y
```

The outer conditional contains two branches. The first branch contains a simple output statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both output statements, although they could have been conditional statements as well. Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can. Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print "x is a positive single digit."
```

The print statement is executed only if we make it past both the conditions, so we can use the and operator:

```
if ((0 < x) and (x < 10)):
    print "x is a positive single digit."
```

Unfortunately, Boo does not provides an alternative syntax that is similar to mathematical notation:

```
if 0 < x < 10:
    print "x is a positive single digit."
```

This condition is semantically the same as the compound boolean expression and the nested conditional. However, Boo will respond with the following error: `ERROR: Operator '<' cannot be used with a left hand side of type 'bool' and a right hand type of 'int'`

## 4.8 The return statement

The `return` statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

```
import System.Math
def printLogarithm(x as int):
    if x <= 0:
        print "Positive numbers only, please."
        return
    result = Log(x)
    print "The log of x is", result
```

The function `printLogarithm` takes a parameter named `x`. The first thing it does is check whether `x` is less than or equal to 0, in which case it displays an error message and then uses `return` to exit the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

Remember that to use a function from the `math` module, you have to import it or spell out its module name in full.

## **4.9 Recursion**

We mentioned that it is legal for one function to call another, and you have seen several examples of that. We neglected to mention that it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do. For example, look at the following function:

```
def countdown(n as int):  
    if n == 0:  
        print "Blastoff!"  
    else:  
        print n  
        countdown(n-1)
```

`countdown` expects the parameter, `n`, to be a positive integer. If `n` is 0, it outputs the word, "Blastoff!" Otherwise, it outputs `n` and then calls a function named `countdown` (itself) passing `n-1` as an argument. What happens if we call this function like this:

```
countdown(3)
```

The execution of `countdown` begins with `n=3`, and since `n` is not 0, it outputs the value 3, and then calls itself...

The execution of `countdown` begins with `n=2`, and since `n` is not 0, it outputs the value 2, and then calls itself...

The execution of `countdown` begins with `n=1`, and since `n` is not 0, it outputs the value 1, and then calls itself...

The execution of `countdown` begins with `n=0`, and since `n` is 0, it outputs the word, "Blastoff!" and then returns.

The `countdown` that got `n=1` returns.

The countdown that got `n=2` returns.

The countdown that got `n=3` returns.

And then you're back in `main` (what a trip). So, the total output looks like this:

```
3
2
1
Blastoff!
```

As a second example, look again at the functions `newLine` and `threeLines`:

```
def newline():
    print

def threeLines():
    newLine()
    newLine()
    newLine()
```

Although these work, they would not be much help if we wanted to output 2 newlines, or 106. A better alternative would be this:

```
def nLines(n as int):
    if n > 0:
        print
        nLines(n-1)
```

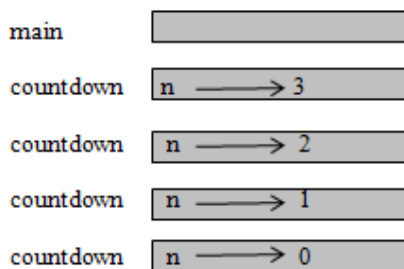
This program is similar to `countdown`; as long as `n` is greater than 0, it outputs one newline and then calls itself to output `n-1` additional newlines. Thus, the total number of newlines is  $1 + (n - 1)$  which, if you do your algebra right, comes out to `n`.



The process of a function calling itself is `recursion`, and such functions are said to be recursive.

## **4.10 Stack diagrams for recursive functions**

In Section 3.11, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.



Every time a function gets called, Boo creates a new function frame, which contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time. This figure shows a stack diagram for `countdown` called with `n = 3`:

As usual, the top of the stack is the frame for `main`. It is empty because we did not create any variables in `main` or pass any parameters to it. The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where `n=0`, is called the base case. It does not make a recursive call, so there are no more frames.

As an exercise, draw a stack diagram for `nLines` called with `n=4`.

## 4.11 Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as infinite recursion, and it is generally not considered a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():
    recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Booish will crash if you call `recurse()` after defining it and you will have to relaunch.

## 4.12 Keyboard input

The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time. Boo provides built-in functions that get input from the keyboard. The simplest is called `prompt`. When this function is called, the program stops and waits for the user to type something. When the user presses Return or the Enter key, the program resumes and `raw input` returns what the user typed as a string. The `prompt` function takes one string parameter which communicates to the user what kind of input is expected:

```
input = prompt("What day is it? ")
#the above outputs: What day is it?
[Type Monday and press Enter]
print input      #outputs: Monday
```

## 4.13 Glossary

*modulus operator*: An operator, denoted with a percent sign (%),

that works on integers and yields the remainder when one number is divided by another.

*boolean expression*: An expression that is either true or false.

*comparison operator*: One of the operators that compares two values: ==, !=, >, <, >=, and <=.

*logical operator*: One of the operators that combines boolean expressions: and, or, and not.

*conditional statement*: A statement that controls the flow of execution depending on some condition.

*condition*: The boolean expression in a conditional statement that determines which branch is executed.

*compound statement*: A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

*block*: A group of consecutive statements with the same indentation.

*body*: The block in a compound statement that follows the header.

*nesting*: One program structure within another, such as a conditional statement inside a branch of another conditional statement.

*recursion*: The process of calling the function that is currently executing.

*base case*: A branch of the conditional statement in a recursive function that does not result in a recursive call.

*infinite recursion*: A function that calls itself recursively without ever reaching the base case. Eventually, an infinite recursion causes a run time error.

*prompt*: A visual cue that tells the user to input data.

## Chapter 5: Fruitful functions

### 5.1 Return values

Some of the built-in functions we have used, such as the math functions, have produced results. Calling the function generates a new value, which we usually assign to a variable or use as part of an expression.

```
e = Exp(1.0)
height = radius * Sin(angle)
```

But so far, none of the functions we have written has returned a value. In this chapter, we are going to write functions that return values, which we will call fruitful functions, for want of a better name. The first example is `area`, which returns the area of a circle with the given radius:

```
import System.Math
def area(radius as int):
    temp = PI * radius**2
    return temp
```

We have seen the `return` statement before, but in a fruitful function the `return` statement includes a return value. This statement means: “Return immediately from this function and use the following expression as a return value.” The expression provided can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius as int):
    return PI * radius**2
```

On the other hand, temporary variables like `temp` often make

debugging easier. Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absoluteValue(x as int):
    if x < 0:
        return -x
    else:
        return x
```

Since these return statements are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called dead code.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absoluteValue(x as int):
    if x < 0:
        return -x
    elif x > 0:
        return x
```

This program is only accidentally correct because if  $x$  happens to be 0, neither condition is true, and the function ends without hitting a return statement. In this case, the return value is the default value of 0:

```
print absoluteValue(0)    #outputs: 0
```

As an exercise, write a compare function that returns 1 if  $x > y$ , 0 if  $x == y$ , and -1 if  $x < y$ .

## **5.2 Program development**

At this point, you should be able to look at complete functions and tell what they do. Also, if you have been doing the exercises, you have written some small functions. As you write larger functions, you might start to have more difficulty, especially with run time and semantic errors.

To deal with increasingly complex programs, we are going to suggest a technique called incremental development<sup>7</sup>. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time. As an example, suppose you want to find the distance between two points, given by the coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$ . By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a distance function should look like in Boo. In other words, what are the inputs (parameters) and what is the output (return value)? In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value.

Already we can write an outline of the function:

```
def distance(x1 as double, y1 as double, x2
as double, y2 as double):
    return 0.0
```

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated. To test the new function, we call it with sample

<sup>7</sup> You do not want to do incremental development by keying in (& re-keying in) lines of code to booish. Instead use the techniques mentioned in section 1.1 (use Notepad & booish's /load command, or better yet SharpDevelop).

values:

```
print distance(1, 2, 4, 6)    #outputs: 0.0
```

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer. At this point we have confirmed that the function is syntactically correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be in the last line we added. A logical first step in the computation is to find the differences  $x_2 - x_1$  and  $y_2 - y_1$ . We will store those values in temporary variables named  $dx$  and  $dy$  and print them.

```
def distance(x1 as double, y1 as double, x2
as double, y2 as double):
    dx as double = x2 - x1
    dy as double = y2 - y1
    print "dx is", dx
    print "dy is", dy
    return 0.0
```

If the function is working, the outputs should be 3 and 4. If so, we know that the function is getting the right parameters and performing the first computation correctly. If not, there are only a few lines to check. Next we compute the sum of squares of  $dx$  and  $dy$ :

```
def distance(x1 as double, y1 as double, x2
as double, y2 as double):
    dx as double = x2 - x1
    dy as double = y2 - y1
    dsquared = dx**2 + dy**2
    print "dsquared is: ", dsquared
    return 0.0
```



Notice that we removed the print statements we wrote in the previous step. Code like that is called scaffolding because it is helpful for building the program but is not part of the final product. Again, we would run the program at this stage and check the output (which should be 25). Finally, if we have imported the math module, we can use the sqrt function to compute and return the result:

```
import System.Math
def distance(x1 as double, y1 as double, x2
as double, y2 as double):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = Sqrt(dsquared)
    return result
```

If that works correctly, you are done. Otherwise, you might want to print the value of result before the return statement. When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, the incremental development process can save you a lot of debugging time. The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
2. Use temporary variables to hold intermediate values so you can output and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

As an exercise, use incremental development to write a function called `hypotenuse` that returns the length of the hypotenuse of a right triangle given the lengths of the two legs as parameters. Record each stage of the incremental development process as you go.

### 5.3 Composition

As you should expect by now, you can call one function from within another. This ability is called composition. As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle. Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we previously defined a function, `distance`, that does that:

```
radius = distance(xc, yc, xp, yp)
```

The second step is to find the area of a circle with that radius and return it:

```
result = area(radius)
return result
```

Wrapping that up in a function, we get:

```
def area2(xc as double, yc as double, xp as
double, yp as double):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

We called this function `area2` to distinguish it from the `area` function defined earlier. There can only be one function with a given name within a given module. If you accidentally give the

function the same name, you will have to type *interpreter.Reset()* on a new line. This will reset Booish. Then you can re-key your program. If you find you make such mistakes frequently (everyone does), acquire an Integrated Development Environment (IDE) or use your favorite text editor to write and save your boo programs.

The temporary variables `radius` and `result` are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def area2(xc, yc, xp, yp):  
    return area(distance(xc, yc, xp, yp))
```

As an exercise, write a function `slope(x1, y1, x2, y2)` that returns the slope of the line through the points `(x1, y1)` and `(x2, y2)`. Then use this function in a function called `intercept(x1, y1, x2, y2)` that returns the y-intercept of the line through the points `(x1, y1)` and `(x2, y2)`.

## **5.4 Boolean functions**

Functions can return boolean values, which is often convenient for hiding complicated tests inside functions. For example:

```
def isDivisible(x as int, y as int):  
    if x % y == 0:  
        return true  
    else:  
        return false
```

The name of this function is `isDivisible`. It is common to give boolean functions names that sound like yes/no questions. `isDivisible` returns either `true` or `false` to indicate whether the `x` is or is not divisible by `y`.

We can make the function more concise by taking advantage of the fact that the condition of the if statement is itself a boolean expression. We can return it directly, avoiding the if statement altogether:

```
def isDivisible(x as int, y as int):
    return x % y == 0
```

This session shows the new function in action:

```
print isDivisible(6, 4) #outputs: False
print isDivisible(6, 3) #outputs: True
```

Boolean functions are often used in conditional statements:

```
if isDivisible(x, y):
    print "x is divisible by y"
else:
    print "x is not divisible by y"
```

It might be tempting to write something like:

```
if isDivisible(x, y) == 1:
```

But the extra comparison is unnecessary.

As an exercise, write a function `isBetween(x, y, z)` that returns true if  $y \leq x \leq z$  or false otherwise.

## 5.5 More recursion

So far, you have only learned a small subset of Boo, but you might be interested to know that this subset is a complete programming language, which means that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you

have learned so far (actually, you would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all). Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis.

If you take a course on the Theory of Computation, you will have a chance to see the proof. To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

frabjuous: An adjective used to describe something that is frabjuous.

If you saw that definition, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function factorial, you might get something like this:

$$\begin{aligned}0! &= 1 \\ n! &= n(n - 1)!\end{aligned}$$

This definition says that the factorial of 0 is 1, and the factorial of any other value,  $n$ , is  $n$  multiplied by the factorial of  $n - 1$ . So  $3!$  is 3 times  $2!$ , which is 2 times  $1!$ , which is 1 times  $0!$ . Putting it all together,  $3!$  equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a Boo program to evaluate it. The first step is to decide what the parameters are for this function. With little effort, you should conclude that factorial takes a single parameter:

```
def factorial(n):
```

If the argument happens to be 0, all we have to do is return 1:

```
def factorial(n as int):
    if n == 0:
        return 1
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of  $(n - 1)$  and then multiply it by  $n$ . Remember, Boo uses type inference. We have been taking advantage of it up until now by not specifying our function's type. However, Boo may not be able to infer the type if your function makes a recursive call. Therefore, we specify the type of function by specifying the type at the end of the first line of the function.

```
def factorial(n as int) as int:
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

The flow of execution for this program is similar to the flow of countdown in Section 4.9. If we call factorial with the value 3:

Since 3 is not 0, we take the second branch and calculate the factorial of  $n-1$ ...

Since 2 is not 0, we take the second branch and calculate the factorial of  $n-1$ ...

Since 1 is not 0, we take the second branch and calculate the factorial of  $n-1$ ...

Since 0 is 0, we take the first branch and return 1 without making any more recursive calls.

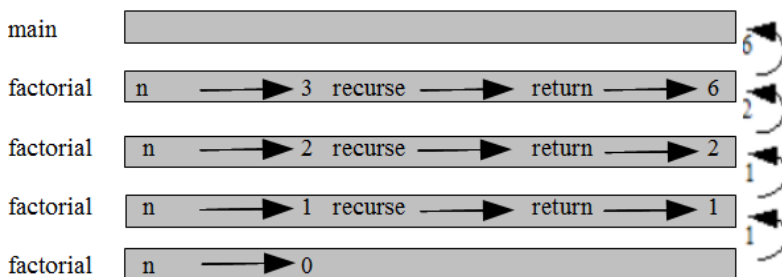
The return value (1) is multiplied by  $n$ , which is 1, and the

result is returned.

The return value (1) is multiplied by n, which is 2, and the result is returned.

The return value (2) is multiplied by n, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

Here is what the stack diagram looks like for this sequence of function calls:



The return values are shown being passed back up the stack. In each frame, the return value is the value of result, which is the product of n and recurse. Notice that in the last frame, the local variables recurse and result do not exist, because the branch that creates them did not execute.

## **5.6 Leap of faith**

Following the flow of execution is one way to read programs, but it can quickly become labyrinthine. An alternative is what we call the “leap of faith.” When you come to a function call, instead of following the flow of execution, you assume that the function works correctly and returns the appropriate value. In fact, you are already practicing this leap of faith when you use built-in functions. When you call `cos` or `exp`, you don't examine the

implementations of those functions. You just assume that they work because the people who wrote the built-in libraries were good programmers.

The same is true when you call one of your own functions. For example, in Section 5.4, we wrote a function called `isDivisible` that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct -- by testing and examining the code -- we can use the function without looking at the code again. The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (yields the correct result) and then ask yourself, "Assuming that I can find the factorial of  $(n - 1)$ , can I compute the factorial of  $n$ ?" In this case, it is clear that you can, by multiplying by  $n$ . Of course, it's a bit strange to assume that the function works correctly when you haven't finished writing it, but that's why it's called a leap of faith!

## 5.7 One more example

In the previous example, we used temporary variables to spell out the steps and to make the code easier to debug, but we could have saved a few lines:

```
def factorial(n as int) as int:
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

From now on, we will tend to use the more concise form, but we recommend that you use the more explicit version while you are developing code. When you have it working, you can tighten it up if you are feeling inspired. After factorial, the most common example of a recursively defined mathematical function is



### *Learning to Program with Boo*

fibonacci, which has the following definition:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2);
```

Translated into Boo, it looks like this:

```
def fibonacci(n as int) as int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

If you try to follow the flow of execution here, even for fairly small values of *n*, your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

```
print fibonacci(0)
print fibonacci(1)
print fibonacci(2)
print fibonacci(3)
print fibonacci(4)
print fibonacci(5)
print fibonacci(6)
print fibonacci(7)
```

Note we had to define the function type as `int` again because of the recursive call within the function.

## **5.8 Glossary**

*fruitful function*: A function that yields a return value.

*return value*: The value provided as the result of a function call.

*temporary variable*: A variable used to store an intermediate value in a complex calculation.

*dead code*: Part of a program that can never be executed, often because it appears after a return statement.

*None*: A special Boo value returned by functions that have no return statement, or a return statement without an argument. Appears as null.

*incremental development*: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

*scaffolding*: Code that is used during program development but is not part of the final version.

## Chapter 6: Iteration

### 6.1 Multiple assignment

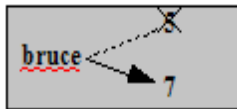
As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
bruce = 5
print bruce
bruce = 7
print bruce
```

The output of this program is:

5  
7

Here is what multiple assignment looks like in a state diagram:



With multiple assignment it is especially important to distinguish between an assignment operation and a statement of equality. Because Boo uses the equal sign (=) for assignment, it is tempting to interpret a statement like  $a = b$  as a statement of equality. It is not!

First, equality is commutative and assignment is not. For example, in mathematics, if  $a = 7$  then  $7 = a$ . But in Boo, the statement  $a = 7$  is legal and  $7 = a$  is not. Furthermore, in mathematics, a statement of equality is always true. If  $a = b$  now, then  $a$  will always equal  $b$ . In Boo, an assignment statement can make two variables equal, but they don't have to stay that way:

```
a = 5
b = a  #a and b are now equal
a = 3  #a and b are no longer equal
```

The third line changes the value of  $a$  but does not change the value of  $b$ , so they are no longer equal. (In some programming languages, a different symbol is used for assignment, such as  $<-$  or  $:=$ , to avoid confusion.) Although multiple assignment is frequently helpful, you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

## 6.2 The while statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. We have seen two programs, `nLines` and `countdown`, that use recursion to perform repetition, which is also called iteration. Because iteration is so common, Boo provides several language features to make it easier. The first feature we are going to look at is the while statement.

Here is what `countdown` looks like with a while statement:

```
def countdown(n as int):
    while n > 0:
        print n
        n = n-1
```

```
print "Blastoff!"
```

Since we removed the recursive call, this function is not recursive. You can almost read the while statement as if it were English. It means, "While *n* is greater than 0, continue displaying the value of *n* and then reducing the value of *n* by 1. When you get to 0, display the word Blastoff!" More formally, here is the flow of execution for a while statement:

1. Evaluate the condition, yielding 0 or 1.
2. If the condition is false (0), exit the while statement and continue execution at the next statement.
3. If the condition is true (1), execute each of the statements in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation. This type of flow is called a loop because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed. The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an infinite loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of countdown, we can prove that the loop terminates because we know that the value of *n* is finite, and we can see that the value of *n* gets smaller each time through the loop, so eventually we have to get to 0. In other cases, it is not so easy to tell:

```
def sequence(n as int):  
    while n != 1:  
        print n
```

```

if n%2 == 0: # n is even
    n = n/2
else: # n is odd
    n = n*3+1

```

The condition for this loop is  $n \neq 1$ , so the loop will continue until  $n$  is 1, which will make the condition false. Each time through the loop, the program outputs the value of  $n$  and then checks whether it is even or odd. If it is even, the value of  $n$  is divided by 2. If it is odd, the value is replaced by  $n*3+1$ . For example, if the starting value (the argument passed to sequence) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1. Since  $n$  sometimes increases and sometimes decreases, there is no obvious proof that  $n$  will ever reach 1, or that the program terminates. For some particular values of  $n$ , we can prove termination. For example, if the starting value is a power of two, then the value of  $n$  will be even each time through the loop until it reaches 1.

Particular values aside, the interesting question is whether we can prove that this program terminates for all values of  $n$ . So far, no one has been able to prove it or disprove it! As an exercise, rewrite the function `nLines` from Section 4.9 using iteration instead of recursion.

## 6.3 Tables

One of the things loops are good for is generating tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors. When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly) but

shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium used to perform floating-point division.

Although a log table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and their logarithms in the right column:

```
import System.Math
x = 1.0
while x < 10.0:
    print x, '\t', Log(x)
    x = x + 1.0
```

The string '\t' represents a tab character. As characters and strings are displayed on the screen, an invisible marker called the cursor keeps track of where the next character will go. After a print statement, the cursor normally goes to the beginning of the next line. The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program:

1.0	0.0
2.0	0.69314718056
3.0	1.09861228867
4.0	1.38629436112
5.0	1.60943791243
6.0	1.79175946923
7.0	1.94591014906
8.0	2.07944154168

9.0     2.19722457734

If these values seem odd, remember that the `log` function uses base *e*. Since powers of two are so important in computer science, we often want to find logarithms with respect to base 2. To do that, we can use the following formula:

$$\log_2 x = \frac{\log_e x}{\log_e 2}$$

Changing the output statement to:

```
print x, '\t', Log(x)/Log(2.0)
```

yields:

```
1.0    0.0
2.0    1.0
3.0    1.58496250072
4.0    2.0
5.0    2.32192809489
6.0    2.58496250072
7.0    2.80735492206
8.0    3.0
9.0    3.16992500144
```

We can see that 1, 2, 4, and 8 are powers of two because their logarithms base 2 are round numbers. If we wanted to find the logarithms of other powers of two, we could modify the program like this:

```
x = 1.0
while x < 100.0:
    print x, '\t', Log(x)/Log(2.0)
    x = x * 2.0
```



Now instead of adding something to  $x$  each time through the loop, which yields an arithmetic sequence, we multiply  $x$  by something, yielding a geometric sequence. The result is:

1.0	0.0
2.0	1.0
4.0	2.0
8.0	3.0
16.0	4.0
32.0	5.0
64.0	6.0

Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column. Logarithm tables may not be useful any more, but for computer scientists, knowing the powers of two is! As an exercise, modify this program so that it outputs the powers of two up to 65,536 (that's  $2^{16}$ ). Print it out and memorize it.

The backslash character in `'\t'` indicates the beginning of an escape sequence. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence `\n` represents a newline. An escape sequence can appear anywhere in a string; in the example, the tab escape sequence is the only thing in the string.

How do you think you represent a backslash in a string? As an exercise, write a single string that produces this output.

## **6.4 Two-dimensional tables**

A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6. A good way to start is to write a loop that prints the multiples of 2, all on one line:

```
import System.Console
i = 1
while i <= 6:
    Write(2*i)
    Write('\t')
    i = i + 1
Write('\n')
```

The first line imports `System.Console`. `System.Console` contains a `Write` as an alternative to `print`. `Write` will not start a newline until you explicitly pass it a `'\n'`. `Print` automatically ends each statement with a newline. The next line initializes a variable named `i`, which acts as a counter or loop variable. As the loop executes, the value of `i` increases from 1 to 6. When `i` is 7, the loop terminates. Each time through the loop, it displays the value of `2*i`, followed by a tab. After the loop completes, the third `Write` statement starts a new line. The output of the program is:

```
2      4      6      8      10     12
```

So far, so good. The next step is to encapsulate and generalize.

## 6.5 Encapsulation and generalization

Encapsulation is the process of wrapping a piece of code in a function, allowing you to take advantage of all the things functions are good for. You have seen two examples of encapsulation: `printParity` in Section 4.5; and `isDivisible` in Section 5.4.

Generalization means taking something specific, such as printing the multiples of 2, and making it more general, such as printing the multiples of any integer. This function encapsulates the

previous loop and generalizes it to print multiples of n:

```
import System.Console
def printMultiples(n as int):
    i = 1
    while i <= 6:
        Write(n*i)
        Write('\t')
        i = i + 1
    Write('\n')
```

To encapsulate, all we had to do was add the `def` line, which declares the name of the function and the parameter list. To generalize, all we had to do was replace the value 2 with the parameter `n`. If we call this function with the argument 2, we get the same output as before.

With the argument 3, the output is:

```
3      6      9      12     15     18
```

With the argument 4, the output is:

```
4      8      12     16     20     24
```

By now you can probably guess how to print a multiplication table by calling `printMultiples` repeatedly with different arguments. In fact, we can use another loop:

```
i = 1
while i <= 6:
    printMultiples(i)
    i = i + 1
```

Notice how similar this loop is to the one inside `printMultiples`. All we did was replace the `Write` statement

with a function call. The output of this program is a multiplication table:

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36

## 6.6 More encapsulation

To demonstrate encapsulation again, let's take the code from the end of Section 6.5 and wrap it up in a function:

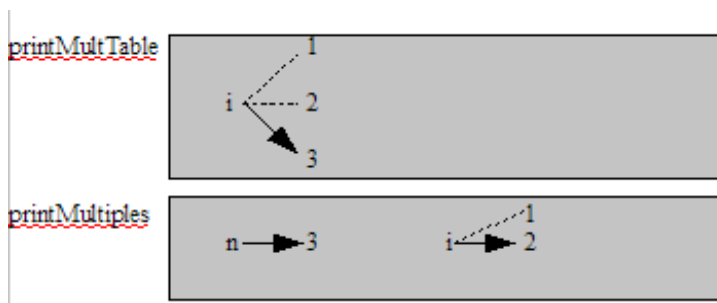
```
def printMultTable():
    i = 1
    while i <= 6:
        printMultiples(i)
        i = i + 1
```

This process is a common development plan. We develop code by writing lines of code outside any function, or typing them in to the interpreter. When we get the code working, we extract it and wrap it up in a function. This development plan is particularly useful if you don't know, when you start writing, how to divide the program into functions. This approach lets you design as you go along.

## 6.7 Local variables

You might be wondering how we can use the same variable, `i`, in both `printMultiples` and `printMultTable`. Doesn't it cause problems when one of the functions changes the value of the variable? The answer is no, because the `i` in

`printMultiples` and the `i` in `printMultTable` are not the same variable. Variables created inside a function definition are local; you can't access a local variable from outside its "home" function. That means you are free to have multiple variables with the same name as long as they are not in the same function. The stack diagram for this program shows that the two variables named `i` are not the same variable. They can refer to different values, and changing one does not affect the other.



The value of `i` in `printMultTable` goes from 1 to 6. In the diagram it happens to be 3. The next time through the loop it will be 4. Each time through the loop, `printMultTable` calls `printMultiples` with the current value of `i` as an argument. That value gets assigned to the parameter `n`. Inside `printMultiples`, the value of `i` goes from 1 to 6. In the diagram, it happens to be 2. Changing this variable has no effect on the value of `i` in `printMultTable`.

It is common and perfectly legal to have local variables in different functions with the same name. In particular, names like `i` and `j` are used frequently as loop variables. If you avoid using them in one function just because you used them somewhere else, you might make the program easier to read.

## 6.8 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the six-by-six table. You could add a parameter to `printMultTable`:

```
def printMultTable(high as int):
    i = 1
    while i <= high:
        printMultiples(i)
        i = i + 1
```

We replaced the value 6 with the parameter `high`. If we call `printMultTable` with the argument 7, it displays:

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36
7	14	21	28	35	42

This is fine, except that we probably want the table to be square with the same number of rows and columns. To do that, we add another parameter to `printMultiples` to specify how many columns the table should have. Just to be annoying, we call this parameter `high`, demonstrating that different functions can have parameters with the same name (just like local variables). Here's the whole program:

```
import System.Console
def printMultiples(n as int, high as int):
    i = 1
    while i <= high:
```

*Learning to Program with Boo*

```
    Write(n*i)
    Write('\t')
    i = i + 1
Write('\n')
def printMultTable(high as int):
    i = 1
    while i <= high:
        printMultiples(i, high)
        i = i + 1
printMultTable(7)
```

Notice that when we added a new parameter, we had to change the first line of the function (the function heading), and we also had to change the place where the function is called in `printMultTable`. As expected, this program generates a square seven-by-seven table:

1	2	3	4	5	6	7
2	4	6	8	10	12	14
3	6	9	12	15	18	21
4	8	12	16	20	24	28
5	10	15	20	25	30	35
6	12	18	24	30	36	42
7	14	21	28	35	42	49

When you generalize a function appropriately, you often get a program with capabilities you didn't plan. For example, you might notice that, because  $ab = ba$ , all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `printMultTable`.

Change `printMultiples(i, high)` to `printMultiples(i, i)` and you get:

```

1
2      4
3      6      9
4      8      12      16
5      10      15      20      25
6      12      18      24      30      36
7      14      21      28      35      42      49

```

As an exercise, trace the execution of this version of `printMultTable` and figure out how it works.

## 6.9 Functions

A few times now, we have mentioned all the things functions are good for. By now, you might be wondering what exactly those things are. Here are some of them:

- Giving a name to a sequence of statements makes your program easier to read and debug.
- Dividing a long program into functions allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.
- Functions facilitate both recursion and iteration.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

## 6.10 Glossary

*multiple assignment*: Making more than one assignment to the same variable during the execution of a program.



*iteration*: Repeated execution of a set of statements using either a recursive function call or a loop.

*loop*: A statement or group of statements that execute repeatedly until a terminating condition is satisfied.

*infinite loop*: A loop in which the terminating condition is never satisfied.

*body*: The statements inside a loop.

*loop variable*: A variable used as part of the terminating condition of a loop.

*tab*: A special character that causes the cursor to move to the next tab stop on the current line.

*newline*: A special character that causes the cursor to move to the beginning of the next line.

*cursor*: An invisible marker that keeps track of where the next character will be printed.

*escape sequence*: An escape character (\) followed by one or more printable characters used to designate a non-printable character.

*encapsulate*: To divide a large complex program into components (like functions) and isolate the components from each other (by using local variables, for example).

*generalize*: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

*development plan*: A process for developing a program. In this chapter, we demonstrated a style of development based on developing code to do simple, specific things and then encapsulating and generalizing.

## Chapter 7: Strings

### 7.1 A compound data type

So far we have seen four types: int, double, boolean, and string. Strings are qualitatively different from the other three because they are made up of smaller pieces, characters. Types that comprise smaller pieces are called compound data types. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful. The bracket operator selects a single character from a string.

```
fruit = "banana"
letter = fruit[1]
print letter
```

The expression `fruit[1]` selects a character from `fruit`. The variable `letter` refers to the result. When we display `letter`, we get a surprise (unless you are a computer scientist):

```
a
```

The first letter of "banana" is not a. You should think of the expression in brackets as an offset from the beginning of the string, and the offset of the first letter is zero. So *b* is the 0<sup>th</sup> letter ("zero-eth") of "banana", *a* is the 1<sup>st</sup> letter ("one-eth"). To get the first letter of a string, you just put 0, or any expression with the value 0, in the brackets:

```
letter = fruit[0]
print letter    #outputs: b
```

The expression in brackets is called an index. An index specifies a member of an ordered set, in this case the set of characters in the

string. The index indicates which one you want, hence the name. It can be any integer expression.

## 7.2 Length

The `len` function returns the number of characters in a string:

```
fruit = "banana"
length = len(fruit)
print length    #outputs: 6
```

To get the last letter of a string, you might be tempted to try something like this:

```
length = len(fruit)
last = fruit[length]
```

```
System.IndexOutOfRangeException: Index was
outside the bounds of the array
```

The reason is that there is no 6th letter in "banana". Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, we have to subtract 1 from `length`:

```
length = len(fruit)
last = fruit[length-1]
```

Alternatively, we can use negative indices<sup>8</sup>, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

## 7.3 Traversal and the for loop

A lot of computations involve processing a string one character at

---

<sup>8</sup> This is not working for strings in Boo 0.9.4.9

a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. One way to encode a traversal is with a while statement:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the `index len(fruit)-1`, which is the last character in the string. As an exercise, write a function that takes a string as an argument and outputs the letters backward, one per line.

Using an index to traverse a set of values is so common that Boo provides an alternative, simpler syntax -- the for loop:

```
for item in fruit:
    print item
```

Each time through the loop, the next item in the string is printed until no items are left. The items in `fruit` are each token in the string.

The following example shows how to use concatenation and a for loop to generate an abecedarian series. "Abecedarian" refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these

names in order:

```
prefixes as string = "JKLMNOPQ"
suffix as string = "ack"
for letter in prefixes:
    print letter + suffix
```

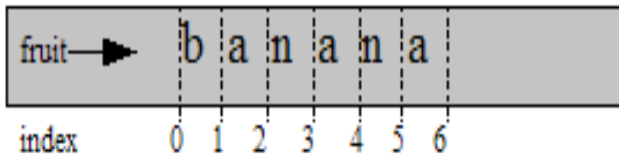
Of course, that's not quite right because “Ouack” and “Quack” are misspelled. As an exercise, modify the program to fix this error.

## 7.4 String slices

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

```
s = "Peter, Paul, and Mary"
print s[0:5]    #outputs: Peter
print s[7:11]   #outputs: Paul
print s[17:21]  #outputs: Mary
```

The operator `[n:m]` returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last. This behavior is counter intuitive; it makes more sense if you imagine the indices pointing between the characters, as in the following diagram:



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

```
fruit = "banana"
print fruit[:3]      #outputs: 'ban'
print fruit[3:]      #outputs: 'ana'
```

What do you think `s[:]` means?

## **7.5 String comparison**

The comparison operators work on strings. To see if two strings are equal:

```
word = "banana"
if word == "banana":
    print "Yes, we have no bananas!"
```

Other comparison operations are useful for putting words in alphabetical order<sup>9</sup>:

```
if word < "banana":
    print "Your word," + word + ", comes
before banana."
elif word > "banana":
    print "Your word," + word + ", comes after
banana."
else:
    print "Yes, we have no bananas!"
```

You should be aware, though, that Boo does not handle upper- and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters. As a result: Your word, Zebra, comes before banana.

A common way to address this problem is to convert strings to a

---

<sup>9</sup> At the time of this writing, an error occurred when the "<" or ">" sign was used to compare Strings.

standard format, such as all lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

## ***7.6 Strings are immutable***

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
greeting = "Hello, world!"
greeting[0] = 'J' # ERROR!
print greeting
```

Instead of producing the output `Jello, world!`, this code causes the compile time error `Error: Property 'string.Chars' is read only`.

Strings are immutable, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print newGreeting
```

The solution here is to concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

## ***7.7 An IndexOf function***

What does the following function do?

```
def IndexOf(str as string, ch as char):
    index = 0
```



```
while index < len(str):
    if str[index] == ch:
        return index
    index = index + 1
return -1
```

In a sense, `IndexOf` is the opposite of the `[]` operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1. This is the first example we have seen of a return statement inside a loop. If `str[index] == ch`, the function returns immediately, breaking out of the loop prematurely. If the character doesn't appear in the string, then the program exits the loop normally and returns -1. This pattern of computation is sometimes called a "eureka" traversal because as soon as we find what we are looking for, we can cry "Eureka!" and stop looking.

Did you have trouble calling `IndexOf`? The `char` type requires special handling so that `boo` can distinguish it from `string`. Try wrapping the character you wish to use inside of a `char` function. For example:

```
print IndexOf('banana', char('a'))
```

As an exercise, modify the `IndexOf` function so that it takes a third parameter, the index in the string where it should start looking.

## **7.8 Looping and counting**

The following program counts the number of times the letter `a` appears in a string:

```
fruit = "banana"
count = 0
```

```

for charact in fruit:
    if charact == char('a'):
        count = count + 1
print count

```

This program demonstrates another pattern of computation called a counter. The variable `count` is initialized to 0 and then incremented each time an `a` is found. (To increment is to increase by one; it is the opposite of decrement, and unrelated to “excrement,” which is a noun.) When the loop exits, `count` contains the result—the total number of `a`'s. As an exercise, encapsulate this code in a function named `countLetters`, and generalize it so that it accepts the string and the letter as parameters.

As a second exercise, rewrite this function so that instead of traversing the string, it uses the three-parameter version of `find` from the previous exercise.

## 7.9 A string type's methods

The string type contains many useful methods. The string type includes a method named `IndexOf` that does the same thing as the function we wrote. To call it we have to specify the name of the variable and the name of the method using dot notation.

```

fruit = "banana"
index = fruit.IndexOf("a")
print index      #outputs: 1

```

This example demonstrates one of the benefits of methods. They help avoid collisions between the names of built-in functions and user-defined functions. By using dot notation we can specify which version of `IndexOf` we want. Actually, `IndexOf` is more general than our version. First, it can find substrings, not just characters. Also, it takes an additional argument that specifies the

index it should start at:

```
print fruit.IndexOf("na", 3) #outputs: 4
```

Or it can take an additional arguments that specifies the number of characters after the start character it should search:

```
print "bob".IndexOf("b", 1, 2) #outputs: 2
```

In this example, if you pass in ("b", 1, 1) the search fails because the letter b does not appear in the index range from 1 to 1.

## **7.10 Character classification**

It is often helpful to examine a character and test whether it is upper- or lower- case, or whether it is a character or a digit. Character methods are useful for these purposes. .Net provides a series of methods for the char type to perform these tests. So we can iterate through a string and fire each characters' method to test if the entire string is upper- or lower- case, or whether it is a character or a digit.

```
def isLower(ch as char):  
    return char.IsLower(ch)
```

Boo will treat individual characters as strings. So to test the function, we need to explicitly declare the value as a char:

```
one = char('A')  
print isLower(one) #outputs: False  
two = char('a')  
print isLower(two) #outputs: True
```

Notice the the right-hand side of the variable assignment is a char function being passed the single-character we want

assigned to our variable on the left-hand side of the assignment.

As yet another alternative, we can use the comparison operator:

```
def isLower(ch as char):
    return (char('a') <= ch and (ch <= char('z')))
```

If `ch` is greater than or equal to 'a' and `ch` is less than or equal to 'z' then `ch` must be lowercase. The function evaluates as True if `ch` is between a and 'z' (inclusive).

As an exercise, discuss which version of `isLower` you think will be fastest. Can you think of other reasons besides speed to prefer one or the other?

As a second homework assignment surf the Microsoft Development Network Library online. This book is available at <http://msdn.com> by selecting the link to '.Net Framework'. Peruse first! Then, as an exercise, find information about the `IsLower` member of the `Char` class we used in this chapter. Clue: it is in the Reference section of the .Net SDK.

## 7.11 Glossary

*item*: a distinct part of a group that can be enumerated in a list.

*compound data type*: A data type in which the values are made up of components, or elements, that are themselves values.

*traverse*: To iterate through the elements of a set, performing a similar operation on each.

*index*: A variable or value used to select a member of an ordered set, such as a character from a string.

*slice*: A part of a string specialized by a range of indices.

*mutable*: A compound data types whose elements can be assigned new values.

*counter*: A variable used to count something, usually initialized to zero and then incremented.

*increment*: To increase the value of a variable by one.

*decrement*: To decrease the value of a variable by one.

*whitespace*: Any of the characters that move the cursor without printing visible characters.

## Chapter 8: Lists

A list is an ordered set of values, where each value is identified by an index. The values that make up a list are called its elements. Lists are similar to strings, which are ordered sets of characters, except that the elements of a list can have any type. Lists and strings and other things that behave like ordered sets are called sequences.

### 8.1 List values

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([ and ]):

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, a char, and (mirabile dictu) another list:

```
["hello", 2.0, char('a'), [10, 20]]
```

A list within another list is said to be nested.

There is a special list that contains no elements. It is called the empty list, and it is denoted []. We can assign list values to variables or pass lists as parameters to functions.

```
vocabulary = ["ameliorate", "castigate",
"defenestrate"]
numbers = [17, 123]
empty = []
```

```
print vocabulary, numbers, empty
#the above outputs: ['ameliorate',
'castigate', 'defenestrate'] [17, 123] []
```

## **8.2 Accessing elements**

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string is the bracket operator ([]). The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
print numbers          #outputs: [17,123]
numbers[1] = 5
print numbers          #outputs: [17,5]
```

The bracket operator can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the one-eth element of numbers, which used to be 123, is now 5. Any integer expression can be used as an index:

```
print numbers[3-2]    #outputs: 5
```

If you try to read or write an element that does not exist, you get a runtime error:

```
numbers[2] = 5
System.IndexOutOfRangeException: Index was
outside of the bounds of the array.
```

If an index has a negative value, it counts backward from the end of the list:

```
print numbers[-1]     #outputs: 5
print numbers[-2]     #outputs: 17
numbers[-3]
```

`System.IndexOutOfRangeException`: Index was outside of the bounds of the array.

`numbers[-1]` is the last element of the list, `numbers[-2]` is the second to last, and `numbers[-3]` doesn't exist. It is common to use a loop variable as a list index.

```
horsemen = ["war", "famine", "pestilence",
"death"]
i = 0
while i < 4:
    print horsemen[i]
    i = i + 1
```

This while loop counts from 0 to 4. When the loop variable `i` is 4, the condition fails and the loop terminates. So the body of the loop is only executed when `i` is 0, 1, 2, and 3. Each time through the loop, the variable `i` is used as an index into the list, printing the `i`-eth element. This pattern of computation is called a list traversal.

### ***8.3 List length***

The function `len` returns the length of a list. It is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list:

```
horsemen = ["war", "famine", "pestilence",
"death"]
i = 0
while i < len(horsemen):
    print horsemen[i]
    i = i + 1
```



The last time the body of the loop is executed, `i` is `len(horsemen) - 1`, which is the index of the last element. When `i` is equal to `len(horsemen)`, the condition fails and the body is not executed, which is a good thing, because `len(horsemen)` is not a legal index. Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam!', 1, ['Brie', 'Roquefort', 'Pol le  
Veq'], [1, 2, 3]]
```

As an exercise, write a loop that traverses the previous list and prints the length of each element. What happens if you send an integer to `len`?

## **8.4 List membership**

"In" is a boolean operator that tests membership in a sequence. We can use `in` with strings, lists and other sequences:

```
horsemen = ['war', 'famine', 'pestilence',  
            'death']  
print 'pestilence' in horsemen  
#the above outputs: True  
print 'debauchery' in horsemen  
#the above outputs: False
```

Since "pestilence" is a member of the `horsemen` list, the `in` operator returns `true`. Since "debauchery" is not in the list, `in` returns `false`. We can use the `not in` combination with `in` to test whether an element is not a member of a list:

```
print 'debauchery' not in horsemen  
#the above outputs: True
```

The `for` loop we saw in Section 7.3 also works with lists. The

generalized syntax of a for loop is:

```
for VARIABLE in LIST:
    BODY
```

This statement is equivalent to:

```
i = 0
while i < len(LIST):
    VARIABLE = LIST[i]
    BODY
    i = i + 1
```

The for loop is more concise because we can eliminate the loop variable, *i*. Here is the previous loop written with a for loop.

```
for horseman in horsemen:
    print horseman
```

It almost reads like English: "For (every) horseman in (the list of) horsemen, print (the name of the) horseman." Any list expression can be used in a for loop:

```
for number in range(20):
    if number % 2 == 0:
        print number

for fruit in ["banana", "apple", "quince"]:
    print "I like to eat " + fruit + "s!"
```

The first example prints all the even numbers between zero and nineteen. The second example expresses enthusiasm for various fruits.

## **8.6 List operations**

The + operator concatenates lists:

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print c      #outputs: [1, 2, 3, 4, 5, 6]
```

Similarly, the \* operator repeats a list a given number of times:

```
print ([0] * 4)      #generates: [0, 0, 0, 0]
print ([1, 2, 3] * 3)
#above generates:[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

## **8.7 List slices**

The slice operations we saw in Section 7.4 also work on lists:

```
list = ['a', 'b', 'c', 'd', 'e', 'f']
print list[1:3]      #outputs: ['b', 'c']
print list[:4] #outputs:['a', 'b', 'c', 'd']
print list[3:] #outputs: ['d', 'e', 'f']
print list[:]
#abov outputs:['a', 'b', 'c', 'd', 'e', 'f']
```

## **8.8 Lists are mutable**

Unlike strings, lists are mutable, which means we can change their elements. Using the bracket operator on the left side of an assignment, we can update one of the elements:

```
fruit = ["banana", "apple", "quince"]
```

```
fruit[0] = "pear"
fruit[-1] = "orange"
print fruit
#above outputs: ['pear', 'apple', 'orange']
```

With the slice operator we can update several elements at once:<sup>10</sup>

```
list = ['a', 'b', 'c', 'd', 'e', 'f']
list[1:3] = ['x', 'y']
print list
#abov outputs:['a', 'x', 'y', 'd', 'e', 'f']
```

We can also remove elements from a list by assigning the empty list to them:

```
list = ['a', 'b', 'c', 'd', 'e', 'f']
list[1:3] = []
print list      #outputs: ['a', 'd', 'e', 'f']
```

And we can add elements to a list by squeezing them into an empty slice at the desired location:

```
list = ['a', 'd', 'f']
list[1:1] = ['b', 'c']
print list
#above outputs: ['a', 'b', 'c', 'd', 'f']
list[4:4] = ['e']
print list
#above outputs:['a', 'b', 'c', 'd', 'e', 'f']
```

## 8.9 List deletion

Using slices to delete list elements can be awkward, and therefore

---

<sup>10</sup> At the time of this writing, the left-hand slicing implementation is incomplete in boo, <http://boo.codehaus.org/Slicing>

error-prone. Boo provides an alternative that is more readable. The `Remove` method removes an element from a list:

```
a = ['one', 'two', 'three']
a.Remove('one')
print a    #outputs: ['two', 'three']
```

`RemoveAt` removes an element from a list by index.

```
a.RemoveAt(0)
print a    #outputs: ['three']
```

As you might expect, `RemoveAt` handles negative indices and causes an error if the index is out of range. You cannot use a slice as an index for `RemoveAt`.

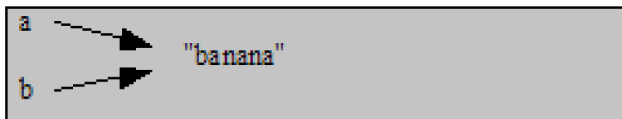
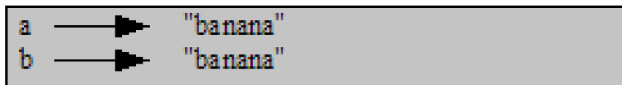
## 8.10 Objects and values

If we execute these assignment statements,

```
a = "banana"
b = "banana"
```

we know that `a` and `b` will refer to a string with the letters "banana". But we can't tell whether they point to the same string.

There are two possible states:



In one case, `a` and `b` refer to two different things that have the same value. In the second case, they refer to the same thing. These “things” have names, they are called objects. An object is something a variable can refer to. Every object has a unique identifier, which we can obtain with the `GetHashCode` method . By printing the identifier of `a` and `b`, we can tell whether they refer to the same object.

```
print a.GetHashCode()
#above outputs something like: -1926346366
print b.GetHashCode()
#above outputs something like: -1926346366
```

## 8.11 Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

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

Because the same list has two different names, `a` and `b`, we say that it is aliased. Changes made to one alias affect the other:

```
b[0] = 5
print a    #outputs: [5, 2, 3]
```

Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects. Of course, for immutable objects, there's no problem. That's why Boo is free to alias strings when it sees an opportunity to economize.

## **8.12 Cloning lists**

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word “copy.” The easiest way to clone a list is to use the slice operator:

```
a = [1, 2, 3]
b = a[:]
print b      #outputs: [1, 2, 3]
```

Taking any slice creates a new list. In this case the slice happens to consist of the whole list. Now we are free to make changes to b without worrying about a:

```
b[0] = 5
print a      #outputs: [1, 2, 3]
```

As an exercise, draw a state diagram for a and b before and after this change.

## **8.13 List parameters**

Passing a list as an argument actually passes a reference to the list, not a copy of the list. For example, the function head takes a list as a parameter and returns the first element:

```
def head(list as List):
    return list[0]
```

Here's how it is used:

```
numbers = [1, 2, 3]
print head(numbers) #outputs: 1
```

If a function modifies a list parameter, the caller sees the change. For example, `deleteHead` removes the first element from a list:

```
def deleteHead(list as List):
    list.RemoveAt(0)
```

Here's how `deleteHead` is used:

```
numbers = [1, 2, 3]
deleteHead(numbers)
print numbers #outputs: [2, 3]
```

If a function returns a list, it returns a reference to the list. For example, `tail` returns a list that contains all but the first element of the given list:

```
def tail(list as List):
    return list[1:]
```

Here's how `tail` is used:

```
numbers = [1, 2, 3]
rest = tail(numbers)
print rest #outputs: [2, 3]
```

Because the return value was created with the slice operator, it is a new list. Creating `rest`, and any subsequent changes to `rest`, have no effect on `numbers`.

## ***8.14 Nested lists***

A nested list is a list that appears as an element in another list. In this list, the three-eth element is a nested list:

```
list = ["hello", 2.0, 5, [10, 20]]
```



If we print `list[3]`, we get `[10, 20]`. To extract an element from the nested list, we can proceed in two steps:

```
elt as List = list[3]
print elt[0]    #outputs: 10
```

Or we can combine them:

```
print list[3][1]    #outputs: 20
```

Bracket operators evaluate from left to right, so this expression gets the three-eth element of `list` and extracts the one-eth element from it.

## **8.15 Matrixes**

Nested lists are often used to represent matrices. For example, the matrix:

```
1 2 3
7 8 9
4 5 6
```

might be represented as:

```
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`matrix` is a list with three elements, where each element is a row of the matrix.

We can select an entire row from the matrix in the usual way:

```
print mat[1]    #outputs: [4, 5, 6]
```

Or we can extract a single element from the matrix using the double-index form:

```
print mat[1][1]          #outputs: 5
```

The first index selects the row, and the second index selects the column. Although this way of representing matrices is common, it is not the only possibility. A small variation is to use a list of columns instead of a list of rows. Later we will see a more radical alternative using a hash.

Not if you try this outside of booish with booi.exe or booc.exe, than you will get an exception, “object does not support slicing”. This is because Duck typing is not enabled by default in those tools. To use the example you will need to type it slightly differently:

```
mat = List[of List]([[1,2,3],[4,5,6],[7,8,9]])
print mat[1][1]
```

## ***8.16 Strings and lists***

Two of the most useful functions in the string module involve lists of strings. The split method breaks a string into a list of words. By default, any number of white space characters is considered a word boundary:

```
song = "The rain in Spain..."
for lyric in song.Split():
    print lyric
#the above outputs on separte lines: 'The
', 'rain', 'in', 'Spain...'
```

An optional argument called a delimiter can be used to specify which characters to use as word boundaries. The following example uses the character a as the delimiter:

### *Learning to Program with Boo*

```
for lyric in song.Split(char('a')):  
    print lyric  
#the above outputs on seperate lines: 'The  
r', 'in in Sp', 'in...'
```

Notice that the delimiter doesn't appear in the list. The join function is the inverse of split. It takes a list of strings and concatenates the elements with a space between each pair:

```
c = ['The', 'rain', 'in', 'Spain...']  
a as string  
a = a.Join(' ', c.ToArray(typeof(string)))  
print a  
#the above outputs: 'The rain in Spain...'
```

Any character you like can be inserted between elements:

```
print a.Join('_', c.ToArray(typeof(string)))  
#the above outputs: 'The_rain_in_Spain...'
```

As an exercise, describe what is happening in `song.Join(' ', song.Split()))`.

## **8.17 Glossary**

*list*: A named collection of objects, where each object is identified by an index.

*index*: An integer variable or value that selects an element of a list.

*element*: One of the values in a list (or other sequence). The bracket operator selects elements of a list.

*sequence*: Any of the data types that consist of an ordered set of elements, with each element identified by an index.

*nested list*: A list that is an element of another list.

*list traversal*: The sequential accessing of each element in a list.

*object*: A thing to which a variable can refer.

*aliases*: Multiple variables that contain references to the same object.

*clone*: To create a new object that has the same value as an existing object. Copying a reference to an object creates an alias but doesn't clone the object.

*delimiter*: A character or string used to indicate where a string should be split.

## **Chapter 9**

### **9.1 This chapter intentionally left blank**

## Chapter 10: Hashes

The compound types you have learned about strings and lists use integers as indices. If you try to use any other type as an index, you get an error. Hashes are similar to other compound types except that they can use any immutable type as an index. As an example, we will create a hash to translate English words into Spanish. For this hash, the indices are strings.

One way to create a hash is to start with the empty hash and add elements. The empty hash is denoted {}:

```
eng2sp = {}
eng2sp['one'] = 'uno'
eng2sp['two'] = 'dos'
```

The first assignment creates a hash named `eng2sp`; the other assignments add new elements to the hash. We can print the current value of the hash:

```
for hashling in eng2sp:
    print hashling.Key
    print hashling.Value
#above outputs: {'one', 'uno', 'two', 'dos'}
on separate lines
```

The elements of a hash appear in a comma-separated list. Each entry contains an index and a value separated by a colon. In a hash, the indices are called keys, so the elements are called key-value pairs. Another way to create a hash is to provide a list of key-value pairs using the syntax:

```
eng2sp = {'one': 'uno', 'two': 'dos',
          'three': 'tres'}
```

If we print the value of `eng2sp` again, we get a surprise:

```
for hashling in eng2sp:
    print(hashling.Key + ':' + hashling.Value)
#above outputs: {'one': 'uno', 'three':
'tres', 'two': 'dos'}
```

The key-value pairs are not in order! Fortunately, there is no reason to care about the order, since the elements of a hash are never indexed with integer indices. Instead, we use the keys to look up the corresponding values:

```
print eng2sp['two']          #outputs: 'dos'
```

The key 'two' yields the value 'dos' even though it appears in the third key-value pair.

## **10.1 Hash operations**

The Remove method deletes a key-value pair from a hash. For example, the following hash contains the names of various fruits and the number of each fruit in stock:

```
inventory = {'apples': 430, 'bananas': 312,
'oranges': 525, 'pears': 217}
```

```
for stuff in inventory:
    print(stuff.Key + ':' + stuff.Value)

#above outputs: {'oranges': 525, 'apples':
430, 'pears': 217, 'bananas': 312}
```

If someone buys all of the pears, we can remove the entry from the hash:

```
inventory.Remove('pears')
for stuff in inventory:
    print(stuff.Key + ':' + stuff.Value)
```

```
#the above outputs: {'oranges': 525,
'apples': 430, 'bananas': 312}
```

Or if we're expecting more pears soon, we might just change the value associated with pears:

```
inventory['pears'] = 0
for stuff in inventory:
    print(stuff.Key + ':' + stuff.Value)
eng2sp = {'one': 'uno', 'two': 'dos',
'three': 'tres'}{'oranges': 525, 'apples':
430, 'pears': 0, 'bananas': 312}
```

The `len` function also works on hashes; it returns the number of key-value pairs:

```
print len(inventory)      #outputs: 4
```

## ***10.2 Hash methods***

The `keys` method takes a hash and returns a list of the keys that appear. A method call is sometimes referred to as an invocation. Empty parens on the end of a method indicate that it takes no parameters.

```
eng2sp = {'one': 'uno', 'two': 'dos',
'three': 'tres'}
for key in eng2sp.Keys:
    print key
#the above outputs: 'one', 'three', 'two'
on separate lines
```

The `values` method takes a hash and returns a list of the values that appear.



```
for value in eng2sp.Values:
    print value
#the above outputs: 'uno', 'tres', 'dos'
on separate lines
```

### **10.3 Aliasing and copying**

Because hashes are mutable, you need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other. If you want to modify a hash and keep a copy of the original, use the copy method. For example, opposites is a hash that contains pairs of opposites:

```
opposites = {'up': 'down', 'right': 'wrong',
            'true': 'false'}
alias = opposites
copy as Boo.Lang.Hash = opposites.Clone()
```

alias and opposites refer to the same object; copy refers to a fresh copy of the same hash. If we modify alias, opposites is also changed:

```
alias['right'] = 'left'
print opposites['right']    #outputs: 'left'
```

If we modify copy, opposites is unchanged:

```
copy['right'] = 'privilege'
print opposites['right']    #outputs: 'left'
```

### **10.4 Sparse matrices**

In Section 8.14, we used a list of lists to represent a matrix. That is a good choice for a matrix with mostly nonzero values, but consider a sparse matrix like this one:

```
0 0 0 1 0
0 0 0 0 0
0 2 0 0 0
0 0 0 0 0
0 0 0 3 0
```

The list representation contains a lot of zeros:

```
matrx = [ [0,0,0,1,0], [0,0,0,0,0],
           [0,2,0,0,0], [0,0,0,0,0], [0,0,0,3,0] ]
```

An alternative is to use a hash. For the keys, we can use tuples that contain the row and column numbers. Here is the hash representation of the same matrix:

```
matrx = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

We only need three key-value pairs, one for each nonzero element of the matrix. Each key is a tuple, and each value is an integer. To access an element of the matrix, we could use the `[]` operator:

```
print matrx[(0,3)] #outputs: 1
```

Notice that the syntax for the hash representation is not the same as the syntax for the nested list representation. Instead of two integer indices, we use one index, which is a tuple of integers. There is one problem. If we specify an element that does not exist, we get a null response because there is no entry in the hash with that key:

```
print matrx[(1,3)] #outputs a blank line
```

The `Contains` method solves this problem.

## Learning to Program with Boo

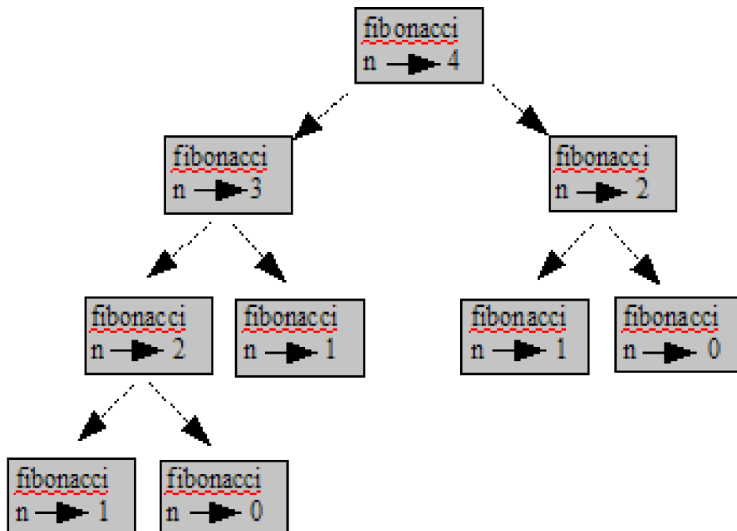
```
if matrX.Contains((0,3)):  
    print matrX[(0,3)]      #outputs: 1  
else:  
    print 0
```

The if clause tests to see if the key contains a value. If true it prints the value, otherwise the else clause will print 0.

### 10.5 Hints

If you played around with the fibonacci function from Section 5.7, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly. On one of our machines, fibonacci(20) finishes instantly, fibonacci(30) takes about a second, and fibonacci(40) takes roughly forever.

To understand why, consider this call graph for fibonacci with n=4:



A call graph shows a set of function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, `fibonacci` with `n=4` calls `fibonacci` with `n=3` and `n=2`. In turn, `fibonacci` with `n=3` calls `fibonacci` with `n=2` and `n=1`. And so on.

Count how many times `fibonacci(0)` and `fibonacci(1)` are called. This is an inefficient solution to the problem, and it gets far worse as the argument gets bigger.

A good solution is to keep track of values that have already been computed by storing them in a hash. A previously computed value that is stored for later use is called a hint. Here is an implementation of `fibonacci` using hints:

```
previous = {0:0, 1:1}
def fibonacci(n as int) as int:
    if previous.ContainsKey(n):
        return previous[n]
    else:
        newValue = fibonacci(n-1)+fibonacci(n-2)
        previous.Add(n, newValue)
        return newValue
print fibonacci(30)           #outputs: 832040
```

The hash named `previous` keeps track of the Fibonacci numbers we already know. We start with only two pairs: 0 maps to 0; and 1 maps to 1. Whenever `fibonacci` is called, it checks the hash to determine if it contains the result. If it's there, the function can return immediately without making any more recursive calls. If not, it has to compute the new value. The new value is added to the hash before the function returns.

Using this version of `fibonacci`, our machines can compute `fibonacci(40)` in an eye blink. But when we try to compute `fibonacci(50)`, we get a different problem:

```
fibonacci(50)
#the above outputs: SystemOverflowException:
Arithmetic operation resulted in an
overflow.
```

The answer, as you will see in a minute, is 1,258,626,025. The problem is that this number is too big to fit into a boo integer. It overflows. Fortunately, this problem has an easy solution.

## **10.6 Long integers**

Boo provides a type called long int that can handle any size integer. The long data type can be used to handle this case.

```
def fibonacci(n as int) as long:

print fibonacci(50) #outputs: 12,586,269,025
```

## **10.7 Counting letters**

In Chapter 7, we wrote a function that counted the number of occurrences of a letter in a string. A more general version of this problem is to form a histogram of the letters in the string, that is, how many times each letter appears. Such a histogram might be useful for compressing a text file. Because different letters appear with different frequencies, we can compress a file by using shorter codes for common letters and longer codes for letters that appear less frequently. Hashes provide an elegant way to generate a histogram<sup>11</sup>:

```
letterCounts = {}
for letter in "Mississippi":
    if letterCounts.Contains(letter):
        letterCounts[letter] =
        (letterCounts[letter] cast int) + 1
```

<sup>11</sup> The cast form in this example requires at least Boo version 0.9.4.3504

```

else:
    letterCounts[letter] = 1
for key in letterCounts.Keys:
    print key, letterCounts[key]
#the above outputs:
M 1
p 2
s 4
i 4

```

We start with an empty hash. For each letter in the string, we find the current count and increment it. The cast is needed to convert the hash value to an integer before incrementing. At the end, the hash contains pairs of letters and their frequencies.

## 10.8 Glossary

*hash*: A collection of key-value pairs that maps from keys to values. The keys can be any immutable type, and the values can be any type.

*key*: A value that is used to look up an entry in a hash.

*key-value pair*: One of the items in a hash.

*method*: A kind of function that is called with a different syntax and invoked “on” an object.

*invoke*: To call a method.

*hint*: Temporary storage of a precomputed value to avoid redundant computation.

*overflow*: A numerical result that is too large to be represented in a numerical format.



## Chapter 11: Files and Exceptions

While a program is running, its data is in memory. When the program ends, or the computer shuts down, data in memory disappears. To store data permanently, you have to put it in a file. Files are usually stored on a hard drive, floppy drive, or CD-ROM.

When there are a large number of files, they are often organized into directories (also called “folders”). Each file is identified by a unique name, or a combination of a file name and a directory name. By reading and writing files, programs can exchange information with each other and generate printable formats like PDF.

Working with files is a lot like working with books. To use a book, you have to open it. When you're done, you have to close it. While the book is open, you can either write in it or read from it. In either case, you know where you are in the book. Most of the time, you read the whole book in its natural order, but you can also skip around.

All of this applies to files as well. To open a file, you specify its name and indicate whether you want to read or write. Opening a file creates a file object. In this example, the variable “e” refers to a new file object.

In .Net the most basic classes for working with files are `StreamReader` and `StreamWriter`. As their names imply, `StreamReader` reads files and `StreamWriter` writes files.

The `StreamWriter` class takes the name of a file as a parameter. It implicitly opens the file. Then its `Write` method writes a line to a file. After reading or writing a file, the `Close`



method must be called explicitly. Closing the file tells the system that we are done writing and makes the file available for reading:

```
import System.IO
e = StreamWriter("test.dat")
e.WriteLine("This is the first line.")
e.WriteLine("This is the second line.")
e.Close()
```

Now we can open the file again, this time for reading, and read the contents into a string.

```
import System.IO
f = StreamReader("test.dat")
for line in f:
    print line
f.Close()
```

The `for` loop reads each line in the file `f` and prints the line. After the loop completes we close the file.

If we want to read each character in the file, than we use `ReadLine`, but if we want to read one character at a time we use `Read()`. Using `Read()` is tricky because it does not return a `char` type, but rather the `int` value of the char. So to get it back to a string, we have to convert the integer to a `char` and then add the `char` to a string.

```
import System.IO
out as string
x as int
f = StreamReader("test.dat")

while f:
    x = f.Read()
```

```

    if x < 0:
        break
    else:
        out = out + System.Convert.ToChar(x)
f.Close()
print out

```

The `while` loop is new. The `while` loop continues to execute the code in the loop until the condition is no longer true. The `break` statement is also new. Executing it breaks out of the loop; the flow of execution moves to the first statement after the loop. In this example, the `while` loop is infinite because the condition will always evaluate as true. The only way to get out of the loop is to execute `break`, which happens when we read a negative integer, which happens when we get to the end of the data in `f`.

A helpful tip: Constantly adding to the string `out` is inefficient. A modern and more efficient approach is to use the `StringBuilder` to append to the end of the string.

```

import System.IO
import System.Text

out = StringBuilder()
x as int
g = StreamReader("test.dat")

while g:
    x = g.Read()
    if x < 0:
        break
    else:
        out.Append(System.Convert.ToChar(x))
g.Close()
print out

```

The following function copies a file, reading and writing each line. The first argument is the name of the original file; the second is the name of the new file:

```
import System.IO

def copyFile(oldFile as string, newFile as
string):
    f1 = StreamReader(oldFile)
    f2 = StreamWriter(newFile)
    for line in f1.ReadLine():
        f2.WriteLine(line)
    f1.Close()
    f2.Close()
    return
```

## **11.1 Text files**

A text file is a file that contains printable characters and whitespace, organized into lines separated by newline characters. Let's take the lessons that we learned in the beginning of this chapter one step further. The following is an example of a line-processing program. `filterFile` makes a copy of `oldFile`, omitting any lines that begin with `#`:

```
import System.IO

def filterFile(oldFile as string, newFile as
string):
    f1 = StreamReader(oldFile)
    f2 = StreamWriter(newFile)
    for line in f1:
        if line == -1:
            break
        if line[0] == char('#'):
            continue
```

```
f2.WriteLine(line)
f1.Close()
f2.Close()
```

The `continue` statement ends the current iteration of the loop, but continues looping. The flow of execution moves to the top of the loop, checks the conditions, and proceeds accordingly. Thus, if text is the empty string, the loop exits. If the first character of text is a hash mark, the flow of execution goes to the top of the loop. Only if both conditions fail do we copy text into the new file.

## 11.2 Writing variables

So far we have used string concatenation to create dynamic string variables. An alternative, and more professional approach is to use string interpolation. String interpolation substitutes variables into a string rather than using concatenation. To use string interpolation, one simply references the variables inside of the string prefixed by a dollar sign '\$variable,' and wrapped in parentheses '\$(variable + 6)' if the variable is part of an expression.

Here is an example where we set an `int` variable equal to a number and report the number in the output of a print statement:

```
cards = 52
print "There are $cards cards in a standard
playing deck, not $(cards + 4)"
f2 = StreamReader("c:\\temp\\output.txt")
output = f2.ReadLine()
output.GetType()
f2.Close(There are 52 cards in a standard
playing deck, not 56
```

## **11.3 Directories**

When you create a new file by opening it and writing, the new file goes in the current directory (wherever you were when you ran the program). Similarly, when you open a file for reading, Boo looks for it in the current directory. If you want to open a file somewhere else, you have to specify the path to the file, which is the name of the directory (or folder) where the file is located:

```
import System.IO
f1 = StreamReader("c:\\temp\\test.dat")
print f1.ReadLine()
#the above outputs:"This is the first line."
```

This example opens a file named "test.dat" that resides in a directory named "temp", which resides on the drive "c:". You cannot use '\\' in part of a filename as you might be familiar with from using a file explorer. It is a reserved for walking the directory tree. You must use a double back-slash as a single back-slash is the string escape character.

Note: If you are on Unix the '/' is normally used to separate directories.

## **11.4 Serialization**

When you write values to a file, they are written as strings:

```
import System.IO
f1 = StreamWriter("c:\\output.txt")
numbers as int = 26
f1.Write(numbers)
f1.Write(numbers)
f1.Close()
```

The problem is that when you read the value back, you get a string. The original type information has been lost. In fact, you

can't even tell where one value ends and the next begins:

```
import System.IO
f2 = StreamReader("c:\\temp\\output.txt")
output = f2.ReadLine()
print output.GetType()
#the above outputs: System.String
f2.Close()
```

The solution is called **serialization**. **Serialization** preserves data structures. `System.Runtime.Serialization` contains the necessary commands. To use it, we created a class (explained in the next chapter) of `Person` named Eric Idle, then serialized him into buffer. We then deserialized back as `Person p`, and checked if he remembered his name. [This example reads better when copied to a wider media than this ebook provides.]

```
import System.IO
import
System.Runtime.Serialization.Formatters.Bina
ry

class Person:
    [property(Name)] _name as string = ""

def pickle(obj):
    BinaryFormatter().Serialize(buffer=MemoryStr
eam(), obj)
    return buffer.ToArray()

def unpickle(buffer as (byte)):
    return
BinaryFormatter().Deserialize(MemoryStream(b
uffer))
```

```
buffer = pickle(Person(Name: "Eric Idle"))
p as Person = unpickle(buffer)
print p.Name      #outputs: Eric Idle
```

We named the functions we used in this example `pickle` and `unpickle` respectively. Pickling is a canning technique used to preserve vegetables for long periods of time. In this case, we are pickling data. Pickling is also the term used by the Python community when serializing data.

## **11.5 Exceptions**

Whenever a runtime error occurs, it creates an exception. Usually, the program stops and Boo prints an error message. For example, dividing by zero creates an exception:

```
i = 1
print 55/(i - 1)
#the above outputs:
System.DivideByZeroException: Attempted to
divide by zero.
```

Also, accessing a nonexistent list item:

```
a = []
print a[5]
#the above outputs:
System.IndexOutOfRangeException: Index was
outside the bounds of the array.
```

The error message has two parts: the type of error before the colon, and specifics about the error after the colon. Normally Boo also prints a traceback of where the program was, but we have omitted that from the examples. Sometimes we want to execute an operation that could cause an exception, but we don't want the program to stop. We can handle the exception using the `try` and

except statements. For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

```
import System
import System.IO
print("Enter a filename:")
input = Console.ReadLine()
try:
    fl = StreamReader(input)
except:
    print "There is no file named $input"
```

The `try` statement executes the statements in the first block. If no exceptions occur, it ignores the `except` statement. If any exception occurs, it executes the statements in the `except` branch and then continues.

We can encapsulate this capability in a function: `exists` takes a filename and returns true if the file exists, false if it doesn't:

```
import System
import System.IO

def exists(filename as string):
    try:
        fl = StreamReader(filename)
        fl.Close()
        return 1
    except:
        return 0
```

You can use multiple `except` blocks to handle different kinds of exceptions. If your program detects an error condition, you can make it raise an exception. Here is an example that gets input from the user and checks for the value 17. Assuming that 17 is not



valid input for some reason, we raise an exception.

```
import System
import System.IO

def inputNumber():
    Console.WriteLine("Pick a non-0 number")
    try:
        inpX = int.Parse(Console.ReadLine())
        return inpX
    except e as FormatException:
        Console.WriteLine("You must input a
number")

inpNumber as int
while inpNumber == 0:
    inpNumber = inputNumber()

Console.WriteLine ("You picked the number:
$inpNumber")
if inpNumber == 17:
    raise Exception("17 is your unlucky
number")
```

Let's walk through this code example. Structurally, it has two parts. The first is the definition of `inputNumber` and the second is the block of code that collects and evaluates the user input.

The `inputNumber` function calls `Console.WriteLine` to print instructions to the user. Then it reads the keyboard input from the user. There is some exception handling to make sure the user keys an integer.

The next code block creates a variable `inpNumber` as an integer. Then the `while` conditional calls the `inputNumber` function until the user enters a valid integer. The program reports to the

user the value he or she keyed, and finally it raises an exception if the number is 17.

Homework: First, run this program a few times trying to input different values and reviewing the results. Create a flow chart diagramming the various types a user could key-in and what the resulting execution of the code will be.

## **11.6 Glossary**

*file*: A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of bytes..

*directory*: A named collection of files, also called a folder.

*path*: A sequence of directory names that specifies the exact location of a file.

*text file*: A file that contains printable characters organized into lines separated by newline characters.

*break statement*: A statement that causes the flow of execution to exit a loop.

*continue statement*: A statement that causes the current iteration of a loop to end. The flow of execution goes to the top of the loop, evaluates the condition, and proceeds accordingly.

*serialize*: To write a data value in a file along with its type information so that it can be reconstituted later.

*exception*: An error that occurs at runtime.

*handle*: To prevent an exception from terminating a program using the try and except statements.

*Learning to Program with Boo*

*raise*: To signal an exception using the raise statement.

## Chapter 12: Classes and objects

We first encountered the term 'class' in Section 3.4. We created a new class to serialize our data in the previous chapter. In fact we have been using classes all along in each example. Now let's discuss in detail what classes are and how they benefit the computer scientist.

### ***12.1 User-defined compound types***

Having used some of Boo's built-in types, we are ready to create a user-defined type: the Point.

Consider the concept of a mathematical point. In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

A natural way to represent a point in Boo is with two floating-point values. The question, then, is how to group these two values into a compound object. The quick and dirty solution is to use a list or tuple, and for some applications that might be the best choice.

An alternative is to define a new user-defined compound type, also called a class. This approach involves a bit more effort, but it has advantages that will be apparent soon. A class definition looks like this:

```
class Point:
    pass
```

Class definitions can appear anywhere in a program, but they are usually near the beginning (after the import statements). The

syntax rules for a class definition are the same as for other compound statements (see Section 4.4). This definition creates a new class called `Point`. The `pass` statement has no effect; it is only necessary because a compound statement must have something in its body.

By creating the `Point` class, we created a new type, also called `Point`. The members of this type are called instances of the type or objects. Creating a new instance is called instantiation. To instantiate a `Point` object, we call a function named (you guessed it) `Point`:

```
blank = Point()
```

The variable `blank` is assigned a reference to a new `Point` object. A function like `Point` that creates new objects is called a constructor.

## **12.2 Fields and Properties**

We can define the various fields (variables) that define a class. The fields are defined during the class definition phase. By default, fields are invisible to outsiders. The following example defines the `point` class with two properties, `Xcoordinate` & `Ycoordinate`, and two fields, `_xcoordinate` & `_ycoordinate`. The property name is used to access the field from outside the class.

Hint: Remember to reset the interpreter before trying to re-define the class. Booish will throw an Ambiguous reference error if you redefine an existing class.

```
class Point:
    [property(Xcoordinate)] _xcoordinate as
double
    [property(Ycoordinate)] _ycoordinate as
double
```

Now re-instantiate `blank` and add values to the two data points:

```
blank = Point()
blank.Xcoordinate = 3.0
blank.Ycoordinate = 4.0
print blank.Xcoordinate, blank.Ycoordinate
```

This syntax is similar to the syntax for selecting a method from an object, such as `Math.Cos` or `string.Uppercase`. In this case, though, we are selecting a data item from an instance. These named items are called *properties*.

The variable `blank` refers to a `Point` object, which contains two properties. Each property refers to a floating-point number. We can read the value of a property using the same syntax:

```
print blank.Ycoordinate #outputs: 4.0
x = blank.Xcoordinate
print x #outputs: 3.0
```

The expression `blank.Xcoordinate` means, “Go to the object `blank` refers to and get the value of `xcoordinate`.” In this case, we assign that value to a variable named `_xcoordinate`. There is no conflict between the variable `_xcoordinate` and the attribute `Xcoordinate`. The purpose of dot notation is to identify which variable you are referring to unambiguously. You can use dot notation as part of any expression, so the following statements are legal:

```
print '(' + blank.Xcoordinate + ', ' +
blank.Ycoordinate + ')'
distanceSquared = blank.Xcoordinate *
blank.Xcoordinate + blank.Ycoordinate *
blank.Ycoordinate
```

The first line outputs (3.0, 4.0); the second line calculates the value 25.0.

You might be tempted to print the value of blank itself:

```
print blank      #outputs: Point
```

The result indicates that blank is an instance of the Point class. This is probably not the most informative way to display a Point object. You will see how to change it shortly. As an exercise, create and print a Point object, and then use id to print the object's unique identifier.

## **12.3 Instances as parameters**

You can pass an instance as a parameter in the usual way. For example:

```
def printPoint(p as Point):  
    print '(' + p.Xcoordinate + ', ' +  
    p.Ycoordinate + ')'
```

printPoint takes a point as an argument and displays it in the standard format. If you call printPoint(blank), the output is (3, 4).

As an exercise, rewrite the distance function from Section 5.2 so that it takes two Points as parameters instead of four numbers.

## **12.4 Sameness**

The meaning of the word "same" seems perfectly clear until you give it some thought, and then you realize there is more to it than you expected. For example, if you say, "Chris and I have the same car," you mean that his car and yours are the same make and

model, but that they are two different cars.

If you say, “Chris and I have the same mother,” you mean that his mother and yours are the same person. So the idea of “sameness” is different depending on the context.

When you talk about objects, there is a similar ambiguity. For example, if two `Points` are the same, does that mean they contain the same data (coordinates) or that they are actually the same object?

To find out if two references refer to the same object, use the `==` operator. For example:

```
p1 = Point()
p1.Xcoordinate = 3
p1.Ycoordinate = 4
p2 = Point()
p2.Xcoordinate = 3
p2.Ycoordinate = 4
print p1 == p2          #outputs: False
```

Even though `p1` and `p2` contain the same coordinates, they are not the same object. If we assign `p1` to `p2`, then the two variables are aliases of the same object:

```
p2 = p1
print p1 == p2          #outputs: True
```

This type of equality is called shallow equality because it compares only the references, not the contents of the objects.

To compare the contents of the objects -- deep equality -- we can write a function called `samePoint`:

```
def samePoint(p1 as Point, p2 as Point):
```



```
    return (p1.Xcoordinate == p2.Xcoordinate)
and (p1.Ycoordinate == p2.Ycoordinate)
```

Now if we create two different objects that contain the same data, we can use `samePoint` to find out if they represent the same point. Not all languages have the same problem. For example, German has different words for different kinds of sameness. "Same car" in this context would be "gleiche Auto," and "same mother" would be "selbe Mutter."

```
blank = Point()
blank.Xcoordinate = 3.0
blank.Ycoordinate = 4.0
blank2 = Point()
blank2.Xcoordinate = 3.0
blank2.Ycoordinate = 4.0
blank3 = Point()
blank3.Xcoordinate = 3.0
blank3.Ycoordinate = 3.0
print samePoint(blank, blank2)
#the above outputs: True
print samePoint(blank, blank3)
#the above outputs: False
```

Of course, if the two variables refer to the same object, they have both shallow and deep equality.

## **12.5 Rectangles**

Let's say that we want a class to represent a rectangle. The question is, what information do we have to provide in order to specify a rectangle? To keep things simple, assume that the rectangle is oriented either vertically or horizontally, never at an angle.

There are a few possibilities: we could specify the center of the

rectangle (two coordinates) and its size (width and height); or we could specify one of the corners and the size; or we could specify two opposing corners. A conventional choice is to specify the upper-left corner of the rectangle and the size.

Again, we'll define a new class:

```
class Rectangle:
    [property(Width)] _width as double
    [property(Height)] _height as double
```

And instantiate it:

```
box = Rectangle()
box.Width = 100.0
box.Height = 200.0
```

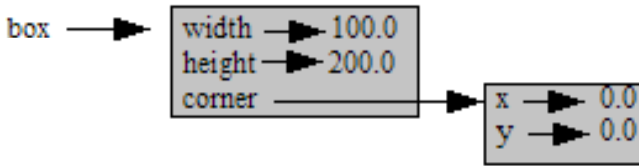
This code redefines the `Rectangle` object with a new property, `Corner`. `Corner` has two floating-point properties. To specify the upper-left corner, we can embed an object within an object!

```
class Rectangle:
    [property(Width)] _width as double
    [property(Height)] _height as double
    [property(Corner)] _corner = Point()

box.Corner.Xcoordinate = 0.0
box.Corner.Ycoordinate = 0.0
```

The dot operator composes. The expression `box.Corner.Xcoordinate` means, “Go to the object `box` refers to and select the property named `corner`; then go to that object and select the property named `Xcoordinate`.”

The figure shows the state of this object:



## **12.6 Instances as return values**

Functions can return instances. For example, `findCenter` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
def findCenter(box as Rectangle):  
    p = Point()  
    p.Xcoordinate = box.Corner.Xcoordinate +  
    box.Width/2.0  
    p.Ycoordinate = box.Corner.Ycoordinate +  
    box.Height/2.0  
    return p
```

To call this function, pass `box` as an argument and assign the result to a variable:

```
center = findCenter(box)  
printPoint(center)  #outputs: (50, 100)
```

## **12.7 Objects are mutable**

We can change the state of an object by making an assignment to one of its properties. For example, to change the size of a rectangle without changing its position, we could modify the values of width and height:

```
box.Width = box.Width + 50
box.Height = box.Height + 100
```

We could encapsulate this code in a method and generalize it to grow the rectangle by any amount:

```
def growRect(box as Rectangle, dwidth as
double, dheight as double):
    box.Width = box.Width + dwidth
    box.Height = box.Height + dheight
```

The variables `dwidth` and `dheight` indicate how much the rectangle should grow in each direction. Invoking this method has the effect of modifying the `Rectangle` that is passed as an argument. For example, we could create a new `Rectangle` named `box1` and pass it to `growRect`:

```
box1 = Rectangle()
box1.Width = 100.0
box1.Height = 200.0
box1.Corner.Xcoordinate = 0.0
box1.Corner.Ycoordinate = 0.0
growRect(box1, 50, 100)
```

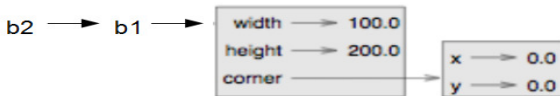
While `growRect` is running, the parameter `box` is an alias for `box1`. Any changes made to `box` also affect `box1`. As an exercise, write a function named `moveRect` that takes a `Rectangle` and two parameters named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the `Xcoordinate` of `Corner` and adding `dy` to the `Ycoordinate` of `Corner`.

## 12.8 Copying

Aliasing can make a program difficult to read because changes made in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object. Consider:

```
b1 = Rectangle()
b1.Width = 10
b1.Height = 10
b1.Corner.Xcoordinate = 1
b1.Corner.Ycoordinate = 2
b2 = b1
```

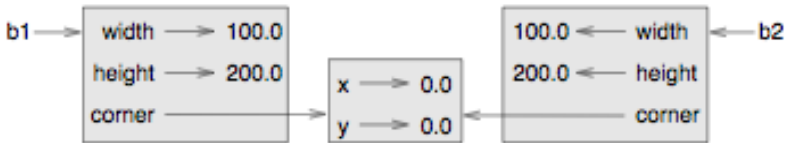
For something like a `Rectangle`, which contains a reference to a `Point`, setting `b2 = b1` creates a reference to `b1`.



What if we want `b2` to be a copy of `b1`. We can use the built-in `MemberwiseClone()`. Consider:

```
b1 = Rectangle()
b2 = Rectangle()
b1.Width = 100
b1.Height = 200
b1.Corner.Xcoordinate = 0
b1.Corner.Ycoordinate = 0
b2 = b1.MemberwiseClone()
b2.Height = 99
print b1.Height      #outputs: 200
print b2.Height      #outputs: 99
```

MemberwiseClone doesn't do quite the right thing. It copies the reference to the `Point` object, so both the old `Rectangle` and the new one refer to a single `Point`. If we create a box, `b1`, in the usual way and then make a copy, `b2`, using `copy`, the resulting state diagram looks like this:



This is almost certainly not what we want. In this case, invoking `growRect` on one of the `Rectangles` would not affect the other, but invoking `moveRect` on either would affect both! This is because the `Height` and `Width` members were copied but the `Point`'s `X` and `Y` coordinates are not copied. Instead, the reference to the point is copied. This behavior is confusing and error-prone. In order to perform a deep copy, you have to implement code for the object class. The code requires an introduction to another .Net concept – Interfaces.

For now, do not worry about what interfaces are. Instead, focus on how to make them work for you. You can use an interface by placing it in the parentheses when you create a new class. For this example, we are going to use the `ICloneable` interface provide by .Net to allow a deep copy:

```
import System

class Point():
```

*Learning to Program with Boo*

```
[property(Xcoordinate)] _xcoordinate as
double
[property(Ycoordinate)] _ycoordinate as
double

class Rectangle(ICloneable):
    [property(Width)] _width as double
    [property(Height)] _height as double
    [property(Corner)] _corner = Point()

    def Clone():
        r = Rectangle(Width, Height, Corner)
        return r

    def constructor():
        pass

    def constructor(width as double, height as
double, corner as Point):
        Width = width
        Height = height
        Corner.Xcoordinate = corner.Xcoordinate
        Corner.Ycoordinate = corner.Ycoordinate
```

Above is the new code to define the Point and Rectangle objects.  
Here is the code to test them:

```
#Test out or new object which can deep copy

box1 = Rectangle() #box1 is a rectangle
box1.Width = 100.0 #set box1's properties
box1.Height = 200.0
box1.Corner.Xcoordinate = 0.5
box1.Corner.Ycoordinate = 0.4

#Copy box two and test for a shallow copy
box2 as Rectangle
```

```

box2 = box1.Clone()
print "The next two lines are false"
print box2 == box1 #Verifies a shallow copy
print box2.Corner == box1.Corner #Verifies a
deep copy
#
#See with your own eyes...
#...that the shallow copy worked
box1.Width=999
print box1.Width      #outputs: 999
print box2.Width      #outputs: 100
#...and that the deep copy worked
box1.Corner.Xcoordinate = 1
box2.Corner.Xcoordinate = 99
print box1.Corner.Xcoordinate #outputs: 1
print box2.Corner.Xcoordinate #outputs: 99

```

Now look back at the `Rectangle` class and study the differences in this implementation of the class and the previous one. First, the line defining the `Rectangle` has `(ICloneable)` specified in parentheses after the class. This alerts the boo compiler that you will be implementing the `ICloneable` interface into the `Rectangle`. The `ICloneable` interface requires you to implement the `Clone()` method inside of any class implementing the `ICloneable` interface (refer to `ICloneable` at MSDN for details). That is the point of the Interface – to force you to implement certain methods, but leave you the flexibility of deciding how it will be done.

Inside of the `Clone()` method, we use a constructor that requires three parameters to constructs a `Rectangle` object. Inside the overloaded constructor, the originating objects `Width`, `Height`, and `Corner` properties are passed to the new, copied instance of the rectangle. Note that corner is not assigned to `corner` in the constructor, but rather that `corner.Xcoordinate` and `corner.Ycoordinate` are set



equal to `Xcoordinate` and `Ycoordinate`. Those lines are where the deep copy actually happens.

The extra constructor method that only contains the `pass` statement allows the original rectangle to be instantiated in the old way.

An an exercise, rewrite `moveRect` so that it creates and returns a new `Rectangle` instead of modifying the old one.

## **12.9 Glossary**

*class*: A user-defined compound type. A class can also be thought of as a template for the objects that are instances of it.

*instantiate*: To create an instance of a class.

*instance*: An object that belongs to a class.

*interface*: A type declaration used to implement abstract methods.

*object*: A compound data type that is often used to model a thing or concept in the real world.

*overload*: A process of specifying the same function or method multiple times to perform similar functions. The functions themselves are differentiated by requiring a unique number of parameters or type of return value.

*constructor*: A method used to create new objects.

*field*: One of the named data items that makes up an instance.

*shallow equality*: Equality of references, or two references that point to the same object.

*deep equality*: Equality of values, or two references that point to objects that have the same value.

*shallow copy*: To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

*deep copy*: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.

## Chapter 13: Classes and functions

### 13.1 Time

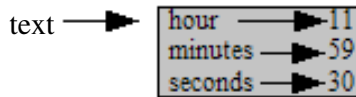
As another example of a user-defined type, we'll define a class called `Time` that records the time of day. The class definition looks like this:

```
class Time():
    [property(Hours)] _hours as int
    [property(Minutes)] _minutes as int
    [property(Seconds)] _seconds as int
```

We can create a new `Time` object and assign properties for hours, minutes, and seconds:

```
time = Time()
time.Hours = 11
time.Minutes = 59
time.Seconds = 30
```

The state diagram for the `Time` object looks like this:



As an exercise, write a method `printTime` for the `Time()` class that prints its properties in the form `hours:minutes:seconds`.

As a second exercise, write a boolean function `after` that takes two `Time` objects, `t1` and `t2`, as arguments, and returns `true` (1) if

`t1` follows `t2` chronologically and `false` (0) otherwise.

## 13.2 Pure functions

In the next few sections, we'll write two versions of a function called `addTime`, which calculates the sum of two `Times`. They will demonstrate two kinds of functions: pure functions and modifiers.

The following is a rough version of `addTime`:

```
def addTime(t1 as Time, t2 as Time):
    sum = Time()
    sum.Hours = t1.Hours + t2.Hours
    sum.Minutes = t1.Minutes + t2.Minutes
    sum.Seconds = t1.Seconds + t2.Seconds
    return sum
```

The function creates a new `Time` object, initializes its properties, and returns a reference to the new object. This is called a pure function because it does not modify any of the objects passed to it as parameters and it has no side effects, such as displaying a value or getting user input.

Here is an example of how to use this function. We'll create two `Time` objects: `currentTime`, which contains the current time; and `breadTime`, which contains the amount of time it takes for a bread-maker to make bread. Then we'll use `addTime` to figure out when the bread will be done. If you haven't finished writing `printTime` yet, take a look ahead to Section 14.2 before you try this:

```
currentTime = Time()
currentTime.Hours = 9
currentTime.Minutes = 14
```

### *Learning to Program with Boo*

```
currentTime.Seconds = 30
breadTime = Time()
breadTime.Hours = 3
breadTime.Minutes = 35
breadTime.Seconds = 0
doneTime = addTime(currentTime, breadTime)
doneTime.printTime()
```

The output of this program is 12:49:30, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one? The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to “carry” the extra seconds into the minutes column or the extra minutes into the hours column.

Here's a second corrected version of the function:

```
def addTime(t1 as Time, t2 as Time):
    sum = Time()
    sum.Hours = t1.Hours + t2.Hours
    sum.Minutes = t1.Minutes + t2.Minutes
    sum.Seconds = t1.Seconds + t2.Seconds
    if sum.Seconds >= 60:
        sum.Seconds = sum.Seconds - 60
        sum.Minutes = sum.Minutes + 1
    if sum.Minutes >= 60:
        sum.Minutes = sum.Minutes - 60
        sum.Hours = sum.Hours + 1
    return sum
```

Although this function is correct, it is starting to get big. Later we will suggest an alternative approach that yields shorter code.

### 13.3 Modifiers

There are times when it is useful for a function to modify one or more of the objects it gets as parameters. Usually, the caller keeps a reference to the objects it passes, so any changes the function makes are visible to the caller. Functions that work this way are called modifiers. `increment`, which adds a given number of seconds to a `Time` object, would be written most naturally as a modifier. A rough draft of the function looks like this:

```
def increment(time as Time, seconds as int):
    time.Seconds = time.Seconds + seconds
    if time.Seconds >= 60:
        time.Seconds = time.Seconds - 60
        time.Minutes = time.Minutes + 1
    if time.Minutes >= 60:
        time.Minutes = time.Minutes - 60
        time.Hours = time.Hours + 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if the parameter `seconds` is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it until `seconds` is less than sixty. One solution is to replace the `if` statements with `while` statements:

```
def increment(time as Time, seconds as int):
    time.Seconds = time.Seconds + seconds
    while time.Seconds >= 60:
        time.Seconds = time.Seconds - 60
        time.Minutes = time.Minutes + 1
    while time.Minutes >= 60:
        time.Minutes = time.Minutes - 60
        time.Hours = time.Hours + 1
```

This function is now correct, but it is not the most efficient

solution. As an exercise, rewrite this function so that it doesn't contain any loops.

As a second exercise, rewrite `increment` as a pure function, and write function calls to both versions.

### **13.4 Which is better?**

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, modifiers are convenient at times, and in some cases, functional programs are less efficient.

In general, we recommend that you write pure functions whenever it is reasonable to do so and resort to modifiers only if there is a compelling advantage. This approach might be called a functional programming style.

In this chapter, we demonstrated an approach to program development that we call prototype development. In each case, we wrote a rough draft (or prototype) that performed the basic calculation and then tested it on a few cases, correcting flaws as we found them. Although this approach can be effective, it can lead to code that is unnecessarily complicated since it deals with many special cases -- and unreliable -- since it is hard to know if you have found all the errors.

An alternative is planned development, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60! The second component is the "ones column," the minute component is the "sixties column," and the hour component is the "thirty-six hundreds column."

When we wrote `addTime` and `increment`, we were effectively doing addition in base 60, which is why we had to carry from one column to the next. This observation suggests another approach to the whole problem, we can convert a `Time` object into a single number and take advantage of the fact that the computer knows how to do arithmetic with numbers. The following function converts a `Time` object into an integer:

```
def convertToSeconds(t as Time):
    totalSeconds as int
    totalSeconds = (t.Hours * 3600) +
(t.Minutes * 60) + t.Seconds
    return totalSeconds
```

Now, all we need is a way to convert from an integer to a `Time` object:

```
def makeTime(seconds as int):
    time = Time()
    time.Hours = seconds/3600
    time.Minutes = (seconds%3600)/60
    time.Seconds = seconds%60
    return time
```

You might have to think a bit to convince yourself that this function is correct. Assuming you are convinced, you can use it and `convertToSeconds` to rewrite `addTime`:

```
def addTime(t1, t2):
    seconds = convertToSeconds(t1) +
convertToSeconds(t2)
    return makeTime(seconds)
```

This version is much shorter than the original, and it is much



easier to demonstrate that it is correct.

As an exercise, rewrite `increment` the same way.

## **13.5 Generalization**

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (`convertToSeconds` and `makeTime`), we get a program that is shorter, easier to read and debug, and more reliable. It is also easier to add features later. For example, imagine subtracting two `Times` to find the duration between them. The native approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

## **13.6 Algorithms**

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written an algorithm. We mentioned this word before but did not define it carefully. It is not easy to define, so we will try a couple of approaches.

First, consider something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100

specific solutions. That kind of knowledge is not algorithmic.

But if you were “lazy,” you probably cheated by learning a few tricks. For example, to find the product of  $n$  and 9, you can write  $n - 1$  as the first digit and  $10 - n$  as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In our opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence. On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming. Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain how we do it, at least not in the form of an algorithm.

## 13.7 Glossary

*pure function*: A function that does not modify any of the objects it receives as parameters. Most pure functions are fruitful.

*modifier*: A function that changes one or more of the objects it receives as parameters. Most modifiers are void.

*functional programming style*: A style of program design in which the majority of functions are pure.

*prototype development*: A way of developing programs starting with a prototype and gradually testing and improving it.

*planned development*: A way of developing programs that involves high-level insight into the problem and more planning than incremental development or prototype development.

*algorithm*: A set of instructions for solving a class of problems by a mechanical, unintelligent process.

## Chapter 14: Classes and methods

### *14.1 Object-oriented features*

Boo is an object-oriented programming language, which means that it provides features that support object-oriented programming. It is not easy to define object-oriented programming, but we have already seen some of its characteristics:

- Programs are made up of object definitions and function definitions, and most of the computation is expressed in terms of operations on objects.
- Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

For example, the `Time` class defined in Chapter 13 corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the `Point` and `Rectangle` classes correspond to the mathematical concepts of a point and a rectangle.

In the `Time` program, there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one `Time` object as a parameter. This observation is the motivation for methods. We have already seen some methods, such as `keys` and `values`, which were invoked on dictionaries. Each method is associated with a class and is intended to be invoked on instances of that class.

Methods are just like functions, with two differences:

- Methods are defined inside a class definition in order to

make the relationship between the class and the method explicit.

- The syntax for invoking a method is different from the syntax for calling a function.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it simply by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

## **14.2 *printTime***

In Chapter 13, we defined a class named `Time` and you wrote a function named `printTime`, which should have looked something like this:

```
class Time:
    pass
def printTime(time as Time):
    print "$(time.Hours):$(time.Minutes):$(
time.Seconds) "
```

To call this function, we passed a `Time` object as a parameter:

```
currentTime = Time()
currentTime.Hours = 9
currentTime.Minutes = 14
currentTime.Seconds = 30
printTime(currentTime)    #outputs: 9:14:30
```

To make `printTime` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation, and dropping the `time` parameter.

```
class Time:
    def printTime():
        print "$Hours:$Minutes:$Seconds"
```

Now we can invoke `printTime` using dot notation.

```
currentTime.printTime() #outputs: 9:14:30
```

The syntax for a function call, `printTime(currentTime)`, suggests that the function is the active agent. It says something like, “Hey `printTime`! Here's an object for you to print.” In object-oriented programming, the objects are the active agents. An invocation like `currentTime.printTime()` says “Hey `currentTime`! Please print yourself!”

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile methods, and makes it easier to maintain and reuse code.

### ***14.3 Another example***

Let's convert `increment` (from Section 13.3) to a method. To save space, we will leave out previously defined methods, but you should keep them in your version:

```
class Time:
    #previous method definitions here...
    def increment(seconds as int):
        Seconds = seconds + Seconds
        while Seconds >= 60:
            Seconds = Seconds - 60
            Minutes = Minutes + 1
        while Minutes >= 60:
            Minutes = Minutes - 60
```

```
Hours = Hours + 1
```

The transformation is purely mechanical. We move the method definition into the class definition and drop the first parameter. Now we can invoke `increment` as a method.

```
currentTime.increment(500)
```

Again, the object on which the method is invoked is `self` aware. The parameter, `seconds` is added to the object's property, `Seconds`. As an exercise, convert `convertToSeconds` (from Section 13.5) to a method in the `Time` class.

## ***14.4 A more complicated example***

The `isAfter` method is slightly more complicated because it operates on two `Time` objects, not just one. The method returns `true` or `false` depending on whether the object's time is after the time provided in the parameter. Naming methods that return boolean as "Is" plus an Adjective is a common coding convention:

```
class Time:
    #previous method definitions here...
    def IsAfter(time2 as Time):
        if Hours > time2.Hours:
            return 1
        if Hours < time2.Hours:
            return 0
        if Minutes > time2.Minutes:
            return 1
        if Minutes < time2.Minutes:
            return 0
        if Seconds > time2.Seconds:
            return 1
```

```
return 0
```

We invoke this method on one object and pass the other as an argument:

```
doneTime = Time()
if doneTime.IsAfter(currentTime):
    print "The bread is not done yet."
else:
    print "The bread must be done."
```

You can almost read the invocation like English: “If the done-time is after the current-time, then...”

## 14.5 Overloading

We have seen built-in functions that take a variable number of arguments. `StreamReader` can take two, three, or four arguments. It is possible to write user-defined functions so they may accept different numbers of parameters. For example, we can upgrade our own version of `IndexOf` to start at any position. We do this by overloading the function. Create additional functions with the additional parameters.

This is the original version from Section 7.7:

```
def indexOf(str as string, ch as char):
    index = 0
    while index < len(str):
        if str[index] == ch:
            return index
        index = index + 1
    return -1
```

This is the new, overloaded version:



### *Learning to Program with Boo*

```
def indexOf(str as string, ch as char, start
as int):
    index = start //default to 0
    while index < len(str):
        if str[index] == ch:
            return index
        index = index + 1
    return -1
```

The third parameter, `start`, is optional because a default value, 0, is used for `index` if `start` is not provided. If we invoke `indexOf` with only two arguments, it uses the zero and starts from the beginning of the string:

```
print indexOf("apple",char('p')) #outputs: 1
```

If we provide a third parameter, it overrides the default:

```
print indexOf ("apple", char('p'), 2)
#the above outputs: 2
print indexOf ("apple", char('p'), 3)
#the above outputs: -1
```

As an exercise, add a fourth parameter, `end`, that specifies where to stop looking.

Warning: This exercise is a bit tricky. The default value of `end` should be `len(str)`, but that doesn't work. The default values are evaluated when the function is defined, not when it is called. When `find` is defined, `str` doesn't exist yet, so you can't find its length.

## **14.6 More on constructors**

We learned in Chapter 12, a constructor is a special method that is

invoked when an object is created. The name of this method is constructor. A constructor method for the `Time` class looks like this:

```
class Time():
    [property(Hours)] _hours as int
    [property(Minutes)] _minutes as int
    [property(Seconds)] _seconds as int

    def constructor(hours as int, minutes as
int, seconds as int):
        Hours = hours
        Minutes = minutes
        Seconds = seconds

    def printTime():
        print "$Hours:$Minutes:$Seconds"
```

There is no conflict between the property `Hours` and the parameter `hours` (Boo variable names are case sensitive). When we invoke the `Time` constructor, the arguments we provide are passed along to the constructor:

```
currentTime = Time(9, 14, 30)
currentTime.printTime() #outputs: 9:14:30
```

## ***14.7 Operator overloading***

Some languages make it possible to change the definition of the built-in operators when they are applied to user-defined types. This feature is called operator overloading. It is especially useful when defining new mathematical types.

For example, to override the addition operator `+`, we modify the `Point` class and the `op_Addition` method:

```
class Point():
    [property(Xcoordinate)] _xcoordinate as
double
    [property(Ycoordinate)] _ycoordinate as
double

    def constructor():
        pass

    def constructor(one as double, two as
double):
        Xcoordinate = one
        Ycoordinate = two

    static def op_Addition(onePoint as Point,
twoPoint as Point) as Point:
        sumPoint = Point()
        sumPoint.Xcoordinate =
onePoint.Xcoordinate + twoPoint.Xcoordinate
        sumPoint.Ycoordinate =
onePoint.Ycoordinate + twoPoint.Ycoordinate
        return sumPoint
```

A lot has changed to the Point class, but let's discuss the `op_Addition` method first. There are two rules for overloading operators. First, the correct method name must be used. In general, the names are prefixed with “`op_`”. A complete list of supported operators may be found in the wiki pages at <http://boo.codehaus.org>. Secondly, the methods must be prefixed with the `static` keyword.

The `op_Addition` method instantiates a new `Point` object that represents the value (by reference) to return when two points are added together. The method performs the math and returns the

point. Notice the `op_Addition` itself is defined as a `Point` in the `def` statement. We have added a new constructor to the `Point` class that takes an `x` and `y` coordinates as inputs which in turn forces us to explicitly declare the null constructor `()` with a `pass` statement. The entire class code is provided in the above example.

Now, when we apply the `+` operator to `Point` objects, `Boo` invokes the new `op_Addition`:

```
aPoint = Point(1,2)
bPoint = Point(3,4)
print aPoint.Xcoordinate      #outputs: 1
print bPoint.Xcoordinate      #outputs: 3
cPoint = aPoint + bPoint
print cPoint.Xcoordinate      #outputs: 4
```

The expression `cPoint = aPoint + bPoint` is equivalent to `cPoint.op_Addition(aPoint, bPoint)`, but obviously more elegant.

As an exercise, add a method `op_Subtraction` that overloads the subtraction operator, and try it out.

## 14.8 Polymorphism

Most of the methods we have written only work for a specific type. When you create a new object, you write methods that operate on that type. Polymorphism not only allows a method to inherit the methods of a parent class, but it allows you to create a related child class that modifies the ways the methods behave.

Consider the `Rectangle` object from chapter 12. What if we added a `GetArea()` method to the `Rectangle` class so that we can easily find the area of a rectangle object:

```
class Rectangle(ICloneable):  
    ...  
    def GetArea():  
        return Width * Height
```

Now we can create a new **Rectangle** and easily retrieve its area:

```
aRectangle = Rectangle()  
aRectangle.Width = 3  
aRectangle.Height = 4  
print aRectangle.GetArea()      #outputs: 12
```

Now let's create a new class, **Square**, that inherits from the **Rectangle** object. We want to modify the **GetArea** method of the **Rectangle** to find **Width \* Width** for the square. Before we create the **Square** class, we have to modify the **Rectangle** method **GetArea** so that **Boo** knows that it is allowed to be overridden. The **virtual** keyword allows a method to be overridden by a child method.

```
class Rectangle(ICloneable):  
    ...  
    virtual def GetArea():  
        return Width * Height
```

The **Square** class will use the **override** keyword to override the virtual method.

```
class Square(Rectangle):  
    override def GetArea():  
        #return area  
        return Width * Width
```

We can instantiate a square and a rectangle object and call the **GetArea** method for both

respectively.

```
#Create aPoint to instantiate the rectangle.
aPoint = Point(Xcoordinate:1,Ycoordinate:2)
bRect = Rectangle(3, 4, aPoint)
print bRect.GetArea()
#the above outputs 12, the area of the
rectangle
cSquare = Square()
cSquare.Width = 5
print cSquare.GetArea()
#above outputs 25, the area of the square
```

Note in the above code we created aPoint and specified its properties as arguments.

## 14.9 Glossary

*object-oriented language*: A language that provides features, such as user-defined classes and inheritance, that facilitate object-oriented programming.

*object-oriented programming*: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

*method*: A function that is defined inside a class definition and is invoked on instances of that class.

*override*: To replace a default. Examples include replacing a default parameter with a particular argument and replacing a default method by providing a new method with the same name.

*initialization method*: A special method (named "constructor") that is invoked automatically when a new object is created and that initializes the object's fields.

*operator overloading*: Extending built-in operators (+, -, \*, >, <, etc.) so that they work with user-defined types.

*dot product*: An operation defined in linear algebra that multiplies two Points and yields a numeric value.

*scalar multiplication*: An operation defined in linear algebra that multiplies each of the coordinates of a Point by a numeric value.

*static*: A method available to the entire class that does not require ownership by a specific instance of the class.

*polymorphic*: A function that can operate on more than one type. If all the operations in a function can be applied to a type, then the function can be applied to a type.

## Chapter 15.0: Sets of objects

### ***15.1 Composition***

By now, you have seen several examples of composition. One of the first examples was using a method invocation as part of an expression. Another example is the nested structure of statements; you can put an if statement within a while loop, within another if statement, and so on. Having seen this pattern, and having learned about lists and objects, you should not be surprised to learn that you can create lists of objects. You can also create objects that contain lists (as fields); you can create lists that contain lists; you can create objects that contain objects; and so on. In this chapter and the next, we will look at some examples of these combinations, using Card objects as an example.

### ***15.2 Card objects***

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, the rank of Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the fields should be: rank and suit. It is not as obvious what type the fields should be. One possibility is to use strings containing words like "Spade" for suits and "Queen" for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to encode the ranks and suits. By



“encode”, we do not mean what some people think, which is to encrypt or translate into a secret code. What a computer scientist means by “encode” is “to define a mapping between a sequence of numbers and the items I want to represent.”

For example:

Spades	→	3
Hearts	→	2
Diamonds	→	1
Clubs	→	0

An obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack	→	11
Queen	→	12
King	→	13

The reason we are using mathematical notation for these mappings is that they are not part of the Boo program. They are part of the program design, but they never appear explicitly in the code. The class definition for the Card type looks like this:

```
class Card():
    _suit as int
    _rank as int
    def constructor(suit as int, rank as int):
        _suit = suit
        _rank = rank
```

As usual, we provide an initialization method that takes an parameter for each field. To create an object that represents the 3 of Clubs, use this command:

```
threeOfClubs = Card(0, 3)
```

The first argument, 0, represents the suit Clubs.

### ***15.3 Class fields and the Show method***

In order to print Card objects in a way that people can easily read, we want to map the integer codes onto words. A natural way to do that is with lists of strings. We assign these lists to class fields at the top of the class definition. We label them static to declare them class fields (not belonging to individual objects), and public (accessible from outside the class):

```
class Card:
    static public suitList as List = ["Clubs",
    "Diamonds", "Hearts", "Spades"]
    static public rankList as List = ["narf",
    "Ace", "2", "3", "4", "5", "6", "7", "8", "9",
    "10", "Jack", "Queen", "King"]
    [property(Suit)] _suit as int
    [property(Rank)] _rank as int

    def constructor(suit as int, rank as int):
        Suit = suit
        Rank = rank
    def Show():
        return "$ (rankList[Rank]) of $
(suitList[Suit]) "
```

A class field is defined outside of any method, and it can be accessed from any of the methods in the class. Inside Show, we can use `suitList` and `rankList` to map the numerical values of Suit and Rank to strings. For example, the expression `suitList[Suit]` means “use the property Suit from the current object as an index into the class field named `suitList`, and select the appropriate string.”

The reason for the "narf" in the first element in `rankList` is to act as a place keeper for the zero-eth element of the list, which will never be used. The only valid ranks are 1 to 13. This wasted item is not entirely necessary. We could have started at 0, as usual, but it is less confusing to encode 2 as 2, 3 as 3, and so on.

With the methods we have so far, we can create and print cards:

```
card1 = Card(1, 11)
print card1.Show() #outputs: Jack of Diamonds
```

Class fields like `suitList` are shared by all `Card` objects. The advantage of this is that we can use any `Card` object to access the class properties:

```
card2 = Card(1, 3)
print card2.suitList[1] #outputs: Diamonds
```

The disadvantage is that if we modify a class field, it affects every instance of the class. For example, if we decide that "Jack of Diamonds" should really be called "Jack of Swirly Whales," we could do this:

```
card1.suitList[1] = "Swirly Whales"
print card1.Show() #outputs: Jack of Swirly
Whales
```

The problem is that all of the Diamonds just became Swirly Whales:

```
print card2.Show() #outputs: 3 of Swirly
Whales
```

It is usually not a good idea to modify class fields.

## 15.4 Comparing cards

For primitive types, there are conditional operators (<, >, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another. For user-defined types, we can override the behavior of the built-in operators by providing a method named `compare`. By convention, `compare` takes two parameters, `self` and `other`, and returns 1 if the first object is greater, -1 if the second object is greater, and 0 if they are equal to each other.

Some types are completely ordered, which means that you can compare any two elements and tell which is bigger. For example, the integers and the floating-point numbers are completely ordered. Some sets are unordered, which means that there is no meaningful way to say that one element is bigger than another. For example, the fruits are unordered, which is why you cannot compare apples and oranges.

The set of playing cards is partially ordered, which means that sometimes you can compare cards and sometimes not. For example, you know that the 3 of Clubs is higher than the 2 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

In order to make cards comparable, you have to decide which is more important, rank or suit. For the sake of choosing, we will say that suit is more important, because a new deck of cards comes sorted with all the Clubs together, followed by all the Diamonds, and so on. With that decided, we can write `compare` :

```
def Compare(acard as Card, mother as Card):
    # check the suits
    if acard.Suit > mother.Suit:
        return 1
    if acard.Suit < mother.Suit:
        return -1
```

```
# suits are the same... check ranks
if acard.Rank < mother.Rank:
    if acard.Rank == 1:
        #Ace trumps
        return 1
    return -1
if acard.Rank > mother.Rank:
    if mother.Rank == 1:
        #Ace trumps
        return -1
    return 1
# ranks are the same... it's a tie
return 0
```

As an exercise rewrite the Compare function so it is a method of the card object that takes one parameter.

## **15.5 Decks**

Now that we have objects to represent Cards, the next logical step is to define a class to represent a Deck. Of course, a deck is made up of cards, so each Deck object will contain a list of cards as an attribute. The following is a class definition for the Deck class. The initialization method creates a list to hold the cards and fills it with the standard set of fifty-two cards:

```
class Deck:
    [property(Cards)] _Card as List
    def constructor():
        Cards = []
        for Suit in range(4):
            for Rank in range(1, 14):
                Cards.Add(Card(Suit, Rank))
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop

enumerates the ranks from 1 to 13. Since the outer loop iterates four times, and the inner loop iterates thirteen times, the total number of times the body is executed is fifty-two (thirteen times four). Each iteration creates a new instance of `Card` with the current suit and rank, and appends that card to the cards list.

## 15.6 *Printing the deck*

As usual, when we define a new type of object we want a method that prints the contents of an object. To print a `Deck`, we traverse the list and print each `Card`'s `Show` method:

```
class Deck:
...
    def PrintDeck():
        for x as Card in Cards:
            print x.Show()
```

Here, and from now on, the ellipsis (...) indicates that we have omitted the other methods in the class. Note the “as `Card`” syntax in the line “for `x as Card` in `Cards`:". This is necessary because `Boo` treats the list as a list of objects. When iterating through the list, the “as `Card`” language instructs `Boo` to treat each object in the list as a `Card`. `Boo` cannot infer the object type.

```
deck = Deck()
deck.PrintDeck()      #outputs a deck of cards
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
```

```
9 of Clubs
10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Diamonds
...
```

## **15.7 Shuffling the deck**

If a deck is perfectly shuffled, then any card is equally likely to appear anywhere in the deck, and any location in the deck is equally likely to contain any card. To shuffle the deck, we will use the `Next` method from the `Random` class. With two integer arguments, `a` and `b`, the overloaded `Random.Next` chooses a random integer in the range  $a \leq x < b$ . Since the upper bound is strictly less than `b`, we can use the length of a list as the second parameter, and we are guaranteed to get a legal index.

The deck's `Shuffle` method will instantiate and instance of `System.Random`. Then will randomly swap two cards 52 times. The number of swaps is somewhat arbitrary. By choosing 52 we illustrate the possibility of setting a maximum boundary for a range to the maximum number of items in a list the the `List.Count` property. Each time we call the `Next` method of the `Random` number class, we give it to parameters, one and fifty-two, this limits the random numbers generated to being within the set of `Cards`.

```
class Deck:
...
    def Shuffle():
        #use random number swap cards
        rndNumber = System.Random()
        #above holds the random number generated
```

```

temp1 = 0
temp2 = 0
for counter in range(1, Cards.Count):
    temp1 = rndNumber.Next(Cards.Count)
    temp2 = rndNumber.Next(Cards.Count)
    #let boo deal with types.
    firstObject = Cards[temp1]
    secondObject = Cards[temp2]
    Cards[temp1] = secondObject
    Cards[temp2] = firstObject

```

A total of 104 cards will be swapped. It is possible that the card will be swapped with itself and therefore the total cards swapped would be less than 104, but that is fine. In fact, if we precluded that possibility, the order of the cards would be less than entirely random

## ***15.8 Removing and dealing cards***

Another method that would be useful for the `Deck` class is `DrawCard`, which takes the next card in the deck and removes it. The `list` method `pop` provides a convenient way to do that:

```

class Deck:
    ...
    def DrawCard():
        if Cards.Count > 1:
            return Cards.Pop()
        else:
            print "No more cards in the deck"

```

## ***15.9 Glossary***

**encode:** To represent one set of values using another set of values by constructing a mapping between them.



class field: A static variable that is defined inside a class definition but outside any method. Class fields are accessible from any method in the class and are shared by all instances of the class.

accumulator: A variable used in a loop to accumulate a series of values, such as by concatenating them onto a string or adding them to a running sum.

## Chapter 16.0: Inheritance

Author's note: Beginning with this chapter, the examples will become progressively more complex. As the examples are not printed in their entirety, you may need to continue reading to find details about any missing methods or functions. Better yet, try to write any missing methods you encounter on your own!

### 16.1 Inheritance

The language feature most often associated with object-oriented programming is inheritance. Inheritance is the ability to define a new class that is a modified version of an existing class.

The primary advantage of this feature is that you can add new methods to a class without modifying the existing class. It is called "inheritance" because the new class inherits all of the methods and properties of the existing class. Extending this metaphor, the existing class is sometimes called the parent class. The new class may be called the child class, or sometimes "subclass."

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the program easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be scattered among several modules. Also, many of the things that can be done using inheritance can be done as elegantly (or more so) without it. If the natural structure of the problem does not lend itself to inheritance,

this style of programming can do more harm than good. In this chapter we will demonstrate the use of inheritance as part of a program that plays the card game Old Maid.

## **16.2 A hand of cards**

For almost any card game, we need to represent a hand of cards. A hand is similar to a deck, of course. Both are made up of a set of cards, and both require operations like adding and removing cards. Also, we might like the ability to shuffle both decks and hands.

A hand is also different from a deck. Depending on the game being played, we might want to perform some operations on hands that don't make sense for a deck. For example, in poker we might classify a hand (straight, flush, etc.) or compare it with another hand. In bridge, we might want to compute a score for a hand in order to make a bid. This situation suggests the use of inheritance. If `Hand` is a subclass of `Deck`, it will have all the methods of `Deck`, and new methods can be added.

In the class definition, the name of the parent class appears in parentheses:

```
class Hand(Deck):  
    pass
```

This statement indicates that the new `Hand` class inherits from the existing `Deck` class.

The `Hand` constructor initializes the fields for the hand, which are `name` and `cards`. The string `name` identifies this hand, probably by the name of the player that holds it. The `name` is an optional parameter with the empty string as a default value. `cards` is the list of cards in the hand, initialized to the empty list:

```

class Hand(Deck):
    [property(Name)] _name as string
    def constructor(name as string):
        Cards = []
        Name = name

    def constructor():
        pass

```

For just about any card game, it is necessary to add and remove cards from the deck. Removing cards is accomplished with a `RemoveCard` method. `RemoveCard` takes a card as an input, finds the card and removes it from the deck. `Draw` is a little bit different. `Draw` takes the top card from the deck (technically the bottom card), removes the card from the deck and returns it to calling method:

```

class Deck:
    ...
    def Draw():
        return Cards.Pop()

    def RemoveCard(input as Card):
        found = Cards.Find({card as Card |
            ((card.Rank == input.Rank) and (card.Suit ==
input.Suit)) })
        if found is not null:
            Cards.Remove(found)
            return 1
        else:
            return 0

```

Again, the ellipsis indicates that we have omitted other methods.

## **16.3 Dealing cards**

Now that we have a `Hand` class, we want to deal cards from the `Deck` into hands. It is not immediately obvious whether this method should go in the `Hand` class or in the `Deck` class, but since it operates on a single deck and (possibly) several hands, it is more natural to put it in `Deck`. `Deal` should be fairly general, since different games will have different requirements. We may want to deal out the entire deck at once or add one card to each hand. `Deal` takes two parameters, a list (or tuple) of hands and the total number of cards to deal. If there are not enough cards in the deck, the method deals out all of the cards and stops:

```
class Deck:
...
    def Deal(hands as List, numberOfCards as int):
        numberOfHands = hands.Count
        for i in range(numberOfCards *
numberOfHands):
            if Cards.Count == 0:
                break #if out of cards
            #who ever is next takes a card
            (hands[i % numberOfHands] as
Hand).Cards.Add(Draw())
```

The loop variable `i` goes from 0 to `numberOfCards` of each player should have in his or her hand times the total number of hands playing. Each time through the loop, a card is removed from the deck using `Draw` method we added which removes and returns the last item in the list.

The modulus operator (%) allows us to deal cards in a round robin (one card at a time to each hand). When it is equal to the number of hands in the list, the expression `i % numberOfHands` wraps around to the beginning of the list (index 0).

Exercise: Write a paragraph explaining how the modulus operator, %, is used to deal the cards. Include a diagram.

## 16.4 Printing a Hand

To print the contents of a hand, we can take advantage of the `printDeck` method inherited from `Deck`. For example:

```
deck = Deck()
deck.Shuffle()
hand = Hand("frank")
deck.Deal([hand], 5)
hand.PrintDeck()      #outputs frank's 5 cards
2 of Spades
3 of Spades
4 of Spades
Ace of Hearts
9 of Clubs
```

It's not a great hand, but it has the makings of a straight flush. Although it is convenient to inherit the existing methods, there are useful shortcuts that we might want in the `Hand` method. When we wanted to print a card with the `Deck` object, we drilled down into the card we wanted and called the card's `ShowCard` method. With a hand of cards we might need to `ShowCards` frequently, so let's create an easier way by encapsulating the call inside a `ShowCard` method for the `Hand` class.

```
class Hand(Deck)
...
    def ShowCard(i as int):
        return cast(Card, Cards[i]).Show()
```

We pass in the index of the card we want, and the card will be returned from the hand.

It may seem odd to send a member of the current `Hand`, to a `Deck` method, until you remember that a `Hand` is a kind of `Deck`. `Hand` objects can do everything `Deck` objects can, so it is legal to send a `Hand` to a `Deck` method. In general, it is always legal to use an instance of a subclass in place of an instance of a parent class.

## **16.5 The Card Game Class**

The `CardGame` class takes care of some basic chores common to all games, such as creating the deck and shuffling it:

```
class CardGame():
    [property (Deck)] _deck as Deck
    def constructor():
        Deck = Deck()
        Deck.Shuffle()
```

This is the first case we have seen where the initialization method performs a significant computation, beyond initializing fields.

To implement specific games, we can inherit from `CardGame` and add features for the new game. As an example, we'll write a simulation of Old Maid.

The object of Old Maid is to get rid of cards in your hand. You do this by matching cards by rank and color. For example, the 4 of Clubs matches the 4 of Spades since both suits are black. The Jack of Hearts matches the Jack of Diamonds since both are red.

To begin the game, the Queen of Clubs is removed from the deck so that the Queen of Spades has no match. The fifty-one remaining cards are dealt to the players in a round robin. After the deal, all players match and discard as many cards as possible.

When no more matches can be made, play begins. In turn, each

player picks a card (without looking) from the closest neighbor to the left who still has cards. If the chosen card matches a card in the player's hand, the pair is removed. Otherwise, the card is added to the player's hand. Eventually all possible matches are made, leaving only the Queen of Spades in the loser's hand.

In our computer simulation of the game, the computer plays all hands. Unfortunately, some nuances of the real game are lost. In a real game, the player with the Old Maid goes to some effort to get their neighbor to pick that card, by displaying it a little more prominently, or perhaps failing to display it more prominently, or even failing to fail to display that card more prominently. The computer simply picks a neighbor's card at random.

A hand for playing Old Maid requires some abilities beyond the general abilities of a `Hand`. We will define a new class, `OldMaidHand`, that inherits from `Hand` and provides two additional methods called `RemoveMatches` and `Compare`. The `Compare` method performs a test of two cards and tests if they are or are not the same. The method is written with a third parameter, `game` that allows an easy path to move the `Compare` method into the deck class at a later time.

Before we create the new class, we will need to add `Color` properties to the `Card` class.

```
class Card:
...
    static public colorList as List = ["Black",
    "Red"]
    [property(Color)] _color as int
...
    def constructor(suit, rank):
        Suit = suit
        Rank = rank
        if Suit == 0 or Suit == 3:
```



*Learning to Program with Boo*

```
        Color = 0
    else:
        Color = 1

    def constructor():
        pass

    def Show():
        return "$(rankList[Rank]) of
$(suitList[Suit]) and is $(colorList[Color])"
...

class OldMaidHand(Hand):
    def constructor(name as string):
        Cards = []
        Name = name

    def Compare(acard as Card, bcard as Card):
        if acard.Color == bcard.Color:
            if acard.Rank == bcard.Rank:
                if acard.Suit != bcard.Suit: #prevents
from comparing the same card
                    return 1 # color and rank match
                return 0

    def RemoveMatches():
        count = 0
        originalCards = Hand()
        originalCards.Cards = Cards[:]
        for card as Card in originalCards.Cards:
            matchCard = Card(card.Suit, card.Rank)
            if card.Suit == 0:
                matchCard.Suit = 3
            if card.Suit == 1:
                matchCard.Suit = 2
            if card.Suit == 2:
                matchCard.Suit = 1
            if card.Suit == 3:
```

```

        matchCard.Suit = 0
        if Cards.Find ( {x as Card | ((x.Rank
== matchCard.Rank) and (x.Suit ==
matchCard.Suit))) ):
            print("The matching pair is...")
            print matchCard.Show()
            print card.Show()
            print("")
            RemoveCard(matchCard)
            RemoveCard(card)
            count = count + 1
    return count

```

With `RemoveMatches`, we start by making a copy of the list of cards, so that we can traverse the copy while removing cards from the hand. Since `Cards` is modified in the loop, we don't want to use it to control the traversal. Boo will not permit you to traverse a list that is changing!

For each card in the hand, we figure out what the matching card is and go looking for it. The match card has the same rank but alternate suite of the same color. If the match card is also in the hand, both cards are removed.

The following example demonstrates how to use `RemoveMatches`:

```

game = CardGame()
hand = OldMaidHand("frank")
game.Deck.Deal([hand], 13)
hand.PrintDeck()
print("=====")
print("remove matches")
print("=====")
hand.RemoveMatches()

```

```
print("Cards remaining in hand...")
hand.PrintDeck()
```

Notice the constructor method for the `OldMaidHand` class. We instantiate the cards list for the hand and we add a `Name` property to assign a player's name to the hand.

## **16.6 OldMaidGame class**

Now we can turn our attention to the game itself. `OldMaidGame` is a subclass of `CardGame` with a new method called `play` that takes a list of players as a parameter. Since `OldMaidGame` inherits from `CardGame`, a new `OldMaidGame` object contains a new shuffled deck:

```
class OldMaidGame(CardGame):
    Players = []

    def Play(names as List):
        # remove Queen of Clubs
        Deck.RemoveCard(Card(0,12))
        # make a hand for each player

        for name in names:
            Players.Add(OldMaidHand(name))
        # deal the cards
        Deck.Deal(Players, 14)
        print "----- Cards have been dealt"
        PrintHands()
        # remove initial matches
        matches as int = RemoveAllMatches()
        print "----- Matches discarded,
play begins"
        PrintHands()

        # play until all 50 cards are matched
```

```

turn = 0
numHands = len(Players)
while matches < 25:
    matches = matches + PlayOneTurn(turn)
    turn = (turn + 1) % numHands
print "who wins?"
print "----- Game is Over"
PrintHands()

```

Some of the steps of the game have been separated into methods. `RemoveAllMatches` traverses the list of hands and invokes `RemoveMatches` on each:

```

class OldMaidGame(CardGame):
...
    def RemoveAllMatches():
        count = 0
        for player as OldMaidHand in Players:
            count = count + player.RemoveMatches()
        return count

```

As an exercise, write `PrintHands` which traverses `Hands` and prints each hand. `Count` is an accumulator that adds up the number of matches in each hand and returns the total. When the total number of matches reaches twenty-five, fifty cards have been removed from the hands, which means that only one card is left and the game is over.

The variable `turn` keeps track of which player's turn it is. It starts at 0 and increases by one each time; when it reaches `NumberOfHands`, the modulus operator wraps it back around to 0. The method `PlayOneTurn` takes a parameter that indicates whose turn it is. The return value is the number of matches made during this turn:

```

class OldMaidGame(CardGame):

```

```
...
def PlayOneTurn(turn as int):
    pickedCard = Card()
    if (Players[turn] as
OldMaidHand).Cards.Count < 1:
        return 0
    neighbor = FindNeighbor(turn)
    #Find the nearest neighbor with cards
    while (Players[neighbor] as
OldMaidHand).Cards.Count < 1:
        neighbor = FindNeighbor(neighbor)
    pickedCard = (Players[neighbor] as
OldMaidHand).Draw()
    (Players[turn] as
OldMaidHand).Cards.Add(Card(pickedCard.Suit,
pickedCard.Rank))
    count = (Players[turn] as
OldMaidHand).RemoveMatches()
    if (Players[turn] as
OldMaidHand).Cards.Count > 1:
        (Players[turn] as
OldMaidHand).Shuffle()
    return count
```

If a player's hand is empty, that player is out of the game, so he or she does nothing and returns 0.

Otherwise, a turn consists of finding the first player on the left that has cards, taking one card from the neighbor, and checking for matches. Before returning, the cards in the hand are shuffled so that the next player's choice is random. The method `FindNeighbor` starts with the player to the immediate left and continues around the circle until it finds a player that still has cards:

```
class OldMaidGame(CardGame):
```

```
...
def FindNeighbor(turn as int) as int:
    neighbor as int
    numHands = len(Players)
    for next in range(1,numHands):
        neighbor = (turn + next) % numHands
    return neighbor
```

The following output<sup>12</sup> is the complete code from the game. As an exercise, modify the code to play with a deck of 11 cards.

```
import System

class Card:
    static public suitList as List = ["Clubs",
    "Diamonds", "Hearts", "Spades"]
    static public rankList as List = ["narf",
    "Ace", "2", "3", "4", "5", "6", "7", "8",
    "9", "10", "Jack", "Queen", "King"]
    static public colorList as List =
    ["Black", "Red"]
    [property(Suit)] _suit as int
    [property(Rank)] _rank as int
    [property(Color)] _color as int

    def constructor(suit as int, rank as int):
        Suit = suit
        Rank = rank
        if Suit == 0 or Suit == 3:
            Color = 0
        else:
            Color = 1

    def constructor():
        pass
```

---

<sup>12</sup> Coloration of Boo types is thanks to SharpDevelop

```
def Show():
    return "$(rankList[Rank]) of $
(suitList[Suit]) and is $(colorList[Color])"

class Deck:
    [property(Cards)] _Card as List

def constructor():
    Cards = []
    for Suit in range(4):
        for Rank in range(1, 14):
            Cards.Add(Card(Suit, Rank))

def PrintDeck():
    for x as Card in Cards:
        print x.Show ()

def Shuffle():
    #use random number swap cards
    rndNumber = System.Random()
    #holds the random number generated
    temp1 = 0
    temp2 = 0
    for counter in range(1, Cards.Count):
        temp1 = rndNumber.Next(Cards.Count)

        temp2 = rndNumber.Next(Cards.Count)
        #let boo deal with types.
        firstObject = Cards[temp1]
        secondObject = Cards[temp2]
        Cards[temp1] = secondObject
        Cards[temp2] = firstObject

def Draw():
    return Cards.Pop()

def RemoveCard(input as Card):
```

```

    found = Cards.Find({card as Card |
    ((card.Rank == input.Rank) and (card.Suit ==
input.Suit)) })
    if found is not null:
        Cards.Remove(found)
        return 1
    else:
        return 0

def Deal(hands as List, numberOfCards as
int):
    numberOfHands = hands.Count
    for i in range(numberOfCards *
numberOfHands):
        if Cards.Count == 0:
            break #if out of cards
            #who ever is next takes a card
            (hands[i % numberOfHands] as
Hand).Cards.Add(Draw())

class Hand(Deck):
    [property(Name)] _name as string

    def constructor(name as string):
        Cards = []
        Name = name

    def constructor():
        pass

    def ShowCard(i as int):
        return cast(Card, Cards[i]).Show()

class CardGame():
    [property (Deck)] _deck as Deck

    def constructor():

```



```
Deck = Deck()
Deck.Shuffle()

class OldMaidHand(Hand):
    def constructor(name as string):
        Cards = []
        Name = name

    def Compare(acard as Card, bcard as Card):
        if acard.Color == bcard.Color:
            if acard.Rank == bcard.Rank:
                if acard.Suit != bcard.Suit:
#prevents from comparing the same card
                    return 1 # color and rank match
                return 0

    def RemoveMatches():
        count = 0
        originalCards = Hand()
        originalCards.Cards = Cards[:]
        for card as Card in originalCards.Cards:
            matchCard = Card(card.Suit, card.Rank)
            if card.Suit == 0:
                matchCard.Suit = 3
            if card.Suit == 1:
                matchCard.Suit = 2
            if card.Suit == 2:
                matchCard.Suit = 1
            if card.Suit == 3:
                matchCard.Suit = 0
            if Cards.Find ( {x as Card | ((x.Rank
== matchCard.Rank) and (x.Suit ==
matchCard.Suit))}):
                print("The matching pair is...")
                print matchCard.Show()
                print card.Show()
                print("")
```

```

        RemoveCard(matchCard)
        RemoveCard(card)
        count = count + 1
    return count

class OldMaidGame(CardGame):
    Players = []

    def FindNeighbor(turn as int) as int:
        neighbor as int
        numHands = len(Players)
        for next in range(1,numHands):
            neighbor = (turn + next) % numHands
        return neighbor

    def PrintHands():
        for player as OldMaidHand in Players:
            print "-----"
            print ((player as OldMaidHand).Name)
            player.PrintDeck()
            print "-----"

    def PlayOneTurn(turn as int):
        pickedCard = Card()
        if (Players[turn] as
OldMaidHand).Cards.Count < 1:
            return 0
        neighbor = FindNeighbor(turn)
        #Find the nearest neighbor with cards
        while (Players[neighbor] as
OldMaidHand).Cards.Count < 1:
            neighbor = FindNeighbor(neighbor)
        pickedCard = (Players[neighbor] as
OldMaidHand).Draw()
        (Players[turn] as
OldMaidHand).Cards.Add(Card(pickedCard.Suit,
pickedCard.Rank))

```

```
        count = (Players[turn] as
OldMaidHand).RemoveMatches()
        if (Players[turn] as
OldMaidHand).Cards.Count > 1:
            (Players[turn] as
OldMaidHand).Shuffle()
        return count

def RemoveAllMatches():
    count = 0
    for player as OldMaidHand in Players:
        count = count + player.RemoveMatches()
    return count

def Play(names as List):
    # remove Queen of Clubs
    Deck.RemoveCard(Card(0,12))
    # make a hand for each player
    for name in names:
        Players.Add(OldMaidHand(name))
    # deal the cards
    Deck.Deal(Players, 14)
    print "----- Cards have been dealt"
    PrintHands()
    # remove initial matches
    matches as int = RemoveAllMatches()
    print "----- Matches discarded,
play begins"
    PrintHands()

    # play until all 50 cards are matched
    turn = 0
    numHands = len(Players)
    while matches < 25:
        matches = matches + PlayOneTurn(turn)
        turn = (turn + 1) % numHands
    print "who wins?"
```

```
print "----- Game is Over"
PrintHands()

game = OldMaidGame()
game.Play(['alva', 'brady', 'charlie', 'diesel'
])

Console.ReadLine()
```

One of our goals is to write code that could be reused to implement other card games. Implement the card game, Go Fish.

## 16.6 Glossary

*inheritance*: The ability to define a new class that is a modified version of a previously defined class.

*parent class*: The class from which a child class inherits.

*child class*: A new class created by inheriting from an existing class; also called a “subclass.”

## Chapter 17.0: Linked lists

### 17.1 *Embedded references*

We have seen examples of fields that refer to other objects, which we called embedded references (see Section 12.8). A common data structure, the linked list, takes advantage of this feature. Linked lists are made up of nodes, where each node contains a reference to the next node in the list. In addition, each node contains a unit of data called the cargo.

A linked list is considered a recursive data structure because it has a recursive definition.

A linked list is either:

- a.) the empty list, represented by None, or
- b.) a node that contains a cargo object and a reference to a linked list.

Recursive data structures lend themselves to recursive methods.

### 17.2 *The Node class*

As usual when writing a new class, we'll start with the initialization methods so that we can test the basic mechanism of creating and displaying the new type:

```
class Node:
    [property(Cargo)] _cargo as Object
    [property(Next)] _next as Node

    def constructor(cargo as object):
        Cargo = cargo
```

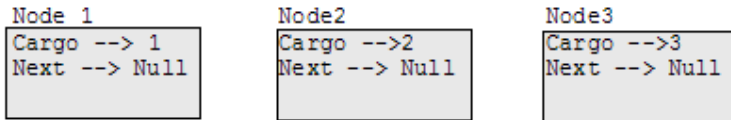
The string representation of a node is just the string representation of the cargo. Since any value can be passed to the `str` function, we can store any value in a list. To test the implementation so far, we can create a `Node` and print it:

```
node = Node("test")
print node.Cargo      #outputs: test
```

To make it interesting, we need a list with more than one node:

```
node1 = Node(1)
node2 = Node(2)
node3 = Node(3)
```

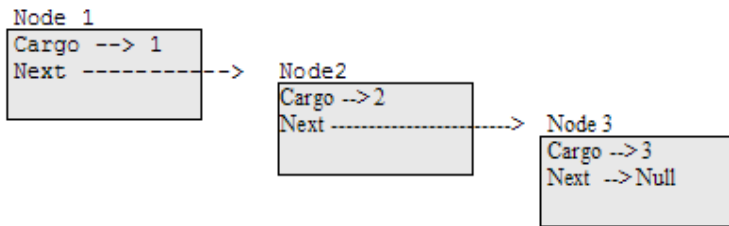
This code creates three nodes, but we don't have a list yet because the nodes are not linked. The state diagram looks like this:



To link the nodes, we have to make the first node refer to the second and the second node refer to the third:

```
node1.Next = node2
node2.Next = node3
```

The reference of the third node is `None`, which indicates that it is the end of the list. Now the state diagram looks like this:



Now you know how to create nodes and link them into lists. What might be less clear at this point is why.

### **17.3 Lists as collections**

Lists are useful because they provide a way to assemble multiple objects into a single entity, sometimes called a collection. In the example, the first node of the list serves as a reference to the entire list.

To pass the list as a parameter, we only have to pass a reference to the first node. For example, the function `printList` takes a single node as an argument. Starting with the head of the list, it prints each node until it gets to the end:

```
def printList(node as Node):
    while node:
        print node.Cargo
        node = node.Next
```

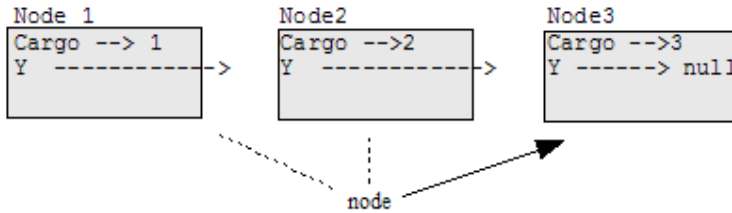
To invoke this method, we pass a reference to the first node:

```
printList(node1)      #outputs: 1 2 3
```

Inside `printList` we have a reference to the first node of the list, but there is no variable that refers to the other nodes. We have to use the `Next` value from each node to get to the next node. To

traverse a linked list, it is common to use a loop variable like `node` to refer to each of the nodes in succession.

This diagram shows the value of `list` and the values that `node` takes on:



By convention, lists are often printed in brackets with commas between the elements, as in `[1, 2, 3]`. As an exercise, modify `printList` so that it generates output in this format.

## 17.4 Lists and recursion

It is natural to express many list operations using recursive methods. For example, the following is a recursive algorithm for printing a list backwards:

1. Separate the list into two pieces: the first node (called the head); and the rest (called the tail).
2. Print the tail backward.
3. Print the head.

Of course, Step 2, the recursive call, assumes that we have a way of printing a list backward. But if we assume that the recursive call works -- the leap of faith -- then we can convince ourselves that this algorithm works. All we need are a base case and a way of proving that for any list, we will eventually get to the base case. Given the recursive definition of a list, a natural base case is



the empty list, represented by Null:

```
def printBackward(list as Node):  
    if list is null:  
        return  
    head = list  
    tail = list.Next  
    printBackward(tail)  
    print head.Cargo.ToString()
```

The first line handles the base case by doing nothing. The next two lines split the list into head and tail. The last two lines print the list.

We invoke this method as we invoked `printList`:

```
printBackward(node1)      #outputs: 3 2 1
```

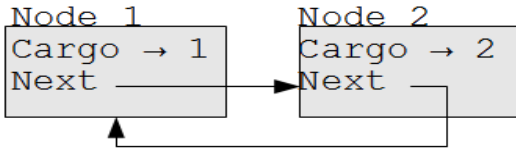
The result is a backward list.

You might wonder why `printList` and `printBackward` are functions and not methods in the `Node` class. The reason is that we want to use null to represent the empty list and it is not legal to invoke a method on null. This limitation makes it awkward to write list-manipulating code in a clean object-oriented style.

Can we prove that `printBackward` will always terminate? In other words, will it always reach the base case? In fact, the answer is no. Some lists will make this method crash.

## **17.5 Infinite lists**

There is nothing to prevent a node from referring back to an earlier node in the list, including itself. For example, this figure shows a list with two nodes, one of which refers to itself:



If we invoke `printList` on this list, it will loop forever. If we invoke `printBackward`, it will recurse infinitely. This sort of behavior makes infinite lists difficult to work with.

Nevertheless, they are occasionally useful. For example, we might represent a number as a list of digits and use an infinite list to represent a repeating fraction. Regardless, it is problematic that we cannot prove that `printList` and `printBackward` terminate. The best we can do is the hypothetical statement, “if the list contains no loops, then these methods will terminate.” This sort of claim is called a precondition. It imposes a constraint on one of the parameters and describes the behavior of the method if the constraint is satisfied. You will see more examples soon.

## 17.6 The fundamental ambiguity problem

One part of `printBackward` might have raised an eyebrow:

```
head = list
tail = list.Next
```

After the first assignment, `head` and `list` have the same type and the same value. So why did we create a new variable?

The reason is that the two variables play different roles. We think of `head` as a reference to a single node, and we think of `list` as a reference to the first node of a list. These “roles” are not part of the program; they are in the mind of the programmer.

In general we can't tell by looking at a program what role a variable plays. This ambiguity can be useful, but it can also make programs difficult to read. We often use variable names like `node` and `list` to document how we intend to use a variable and sometimes create additional variables to disambiguate. We could have written `printBackward` without `head` and `tail`, which makes it more concise but possibly less clear:

```
def printBackward(list as Node):  
    if list is null:  
        return  
    printBackward(list.Next)  
    print list.Cargo.ToString()
```

Looking at the two function calls, we have to remember that `printBackward` treats its argument as a collection and `print` treats its argument as a single object.

The fundamental ambiguity theorem describes the ambiguity that is inherent in a reference to a node:

*A variable that refers to a node might treat the node as a single object or as the first in a list of nodes.*

## **17.7 Modifying lists**

There are two ways to modify a linked list. Obviously, we can change the cargo of one of the nodes, but the more interesting operations are the ones that add, remove, or reorder the nodes. As an example, let's write a method that removes the second node in the list and returns a reference to the removed node:

```
def removeSecond(list as Node):  
    if list == null:  
        return
```

```

first = list
second = list.Next
# make the first node refer to the third
first.Next = second.Next
# separate the second node from the rest
of the list
second.Next = null
return second

```

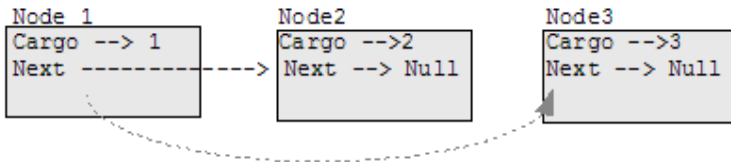
Again, we are using temporary variables to make the code more readable. Here is how to use this method:

```

printList(node1)      #outputs: 1 2 3
removed = removeSecond(node1)
printList(removed)    #outputs: 2
printList(node1)      #outputs: 1 3

```

This state diagram shows the effect of the operation:



What happens if you invoke this method and pass a list with only one element (a singleton)? What happens if you pass the empty list as an argument? Is there a precondition for this method? If so, fix the method to handle a violation of the precondition in a reasonable way.

## 17.8 Wrappers and helpers

It is often useful to divide a list operation into two methods. For example, to print a list backward in the conventional list format

[3, 2, 1] we can use the `printBackward` method to print 3, 2, but we need to revise it to insert the commas. So we will revise `printBackward` to return a string to a second method `printBackwardNicely`. `PrintBackwardNicely` will append the first node, wrap the entire output in brackets and print the list.

```
def printBackward(list as Node) as string:
    out as string
    if list is null:
        return
    head = list
    tail = list.Next
    out = out + printBackward(tail)
    #print head.Cargo.ToString()
    out = out + head.Cargo.ToString() + ', '
    return out

def printBackwardNicely(list as Node):
    finalOut as string
    if list is null:
        return
    else:
        head = list
        tail = list.Next
        finalOut = printBackward(tail)
        print '[' + finalOut +
head.Cargo.ToString() + ']'
```

Again, it is a good idea to check methods like this to see if they work with special cases like an empty list or a singleton. When we use this method elsewhere in the program, we invoke `printBackwardNicely` directly, and it invokes `printBackward` on our behalf. In that sense, `printBackwardNicely` acts as a wrapper, and it uses `printBackward` as a helper.

An examples:

```
node1 = Node(1)
node2 = Node(2)
node3 = Node(3)
node4 = Node(4)
node1.Next = node2
node2.Next = node3
node3.Next = node4
printBackwardNicely(node1)
#the above outputs: [4, 3, 2, 1]
```

## 17.9 The *LinkedList* Class

There are some subtle problems with the way we have been implementing lists. In a reversal of cause and effect, we'll propose an alternative implementation first and then explain what problems it solves.

First, we'll create a new class called `LinkedList`. Its attributes are an integer that contains the length of the list and a reference to the first node. `LinkedList` objects serve as handles for manipulating lists of `Node` objects:

```
class LinkedList:
    [property (Length)] _length as double
    [property (Head)] _head as Node
```

One nice thing about the `LinkedList` class is that it provides a natural place to put wrapper functions like `printBackwardNicely`, which we can make a method of the `LinkedList` class:

```
import System.IO
```

```
...
class LinkedList:
...
    def printBackward():
        Console.Write ("[")
        if Head != null:
            Head.printBackward()
        Console.Write ("]")

class Node:
...
    def printBackward():
        if Next != null:
            tail = Next
            tail.printBackward()
            Console.Write (Cargo)
```

Just to make things confusing, we renamed `printBackwardNicely`. Now there are two methods named `printBackward`: one in the `Node` class (the helper); and one in the `LinkedList` class (the wrapper). When the wrapper invokes `Head.printBackward`, it is invoking the helper, because `Head` is a `Node` object.

Another benefit of the `LinkedList` class is that it makes it easier to add or remove the first element of a list. For example, `addFirst` is a method for `LinkedLists`; it takes an item of cargo as an argument and puts it at the beginning of the list:

```
class LinkedList:
...
    def addFirst(cargo):
        node = Node(cargo)
        node.Next = Head
        Head = node
        Length = Length + 1
```

The class is missing a method to walk to the end of a linked list, and add a node to the end. As usual, you should check code like this to see if it handles the special cases. For example, what happens if the list is initially empty?

## **17.10 Invariants**

Some lists are “well formed”; others are not. For example, if a list contains a loop, it will cause many of our methods to crash, so we might want to require that lists contain no loops. Another requirement is that the length value in the `LinkedList` object should be equal to the actual number of nodes in the list.

Requirements like these are called invariants because, ideally, they should be true of every object all the time. Specifying invariants for objects is a useful programming practice because it makes it easier to prove the correctness of code, check the integrity of data structures, and detect errors.

One thing that is sometimes confusing about invariants is that there are times when they are violated. For example, in `addFirst`, after we have added the node but before we have incremented `length`, the invariant is violated (we should have verified that the length was zero). This kind of violation is acceptable; in fact, it is often impossible to modify an object without violating an invariant for at least a little while.

Normally, we require that every method that violates an invariant must restore the invariant. If there is any significant stretch of code in which the invariant is violated, it is important for the comments to make that clear, so that no operations are performed that depend on the invariant.



## **17.11 Glossary**

*embedded reference*: A reference stored in the field of an object.

*linked list*: A data structure that implements a collection using a sequence of linked nodes.

*node*: An element of a list, usually implemented as an object that contains a reference to another object of the same type.

*cargo*: An item of data contained in a node.

*link*: An embedded reference used to link one object to another.

*precondition*: An assertion that must be true in order for a method to work correctly.

*fundamental ambiguity theorem*: A reference to a list node can be treated as a single object or as the first in a list of nodes.

*singleton*: A linked list with a single node.

*wrapper*: A method that acts as a middleman between a caller and a helper method, often making the method easier or less error-prone to invoke.

*helper*: A method that is not invoked directly by a caller but is used by another method to perform part of an operation.

*invariant*: An assertion that should be true of an object at all times (except perhaps while the object is being modified).

## Chapter 18.0: Stacks

### ***18.1 Abstract data types***

The data types you have seen so far are all concrete, in the sense that we have completely specified how they are implemented. For example, the `Card` class represents a card using two integers. As we discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An abstract data type, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do), but it does not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

- It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.
- Since there are usually many ways to implement an ADT, it might be useful to write an algorithm that can be used with any of the possible implementations.
- Well-known ADTs, such as the `Stack` ADT in this chapter, are often implemented in standard libraries so they can be written once and used by many programmers.
- The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish the code that uses the ADT, called the client code, from the code that implements the ADT, called the provider code.

### ***18.2 The Stack ADT***

In this chapter, we will look at one common ADT, the stack. A

stack is a collection, meaning that it is a data structure that contains multiple elements. Other collections we have seen include dictionaries and lists. An ADT is defined by the operations that can be performed on it, which is called an interface. The interface for a stack consists of these operations:

*init*: Initialize a new empty stack.

*push*: Add a new item to the stack.

*pop*: Remove and return an item from the stack. The item that is returned is always the last one that was added.

*isEmpty*: Check whether the stack is empty.

A stack is sometimes called a “last in, first out” or LIFO data structure, because the last item added is the first to be removed.

### ***18.3 Implementing stacks with boo lists***

The list operations that boo provides are similar to the operations that define a stack. The interface isn't exactly what it is supposed to be, but we can write code to translate from the Stack ADT to the built-in operations. This code is called an implementation of the Stack ADT. In general, an implementation is a set of methods that satisfy the syntactic and semantic requirements of an interface.

To ease the implementation we will name our class `Stack2` so as not to confuse with the `.NET System.Collections.Stack`. Here is an implementation of the Stack ADT that uses a boo list:

```
class Stack2:
    [property(Items)] _items = []
    def Push(item):
        Items.Add(item)
```

```
def Pop():
    return Items.Pop()
def IsEmpty():
    return (Items == [])
```

A `Stack2` object contains an property named `Items` that is a list of items in the stack. Instantiating a `Stack2` sets `Items` to the empty list. To push a new item onto the stack, `push` appends it onto `items`. To pop an item off the stack, `pop` uses the list method to remove and return the last item on the list.

Finally, to check if the stack is empty, `IsEmpty` compares `items` to the empty list.

An implementation like this, in which the methods consist of simple invocations of existing methods, is called a veneer. In real life, veneer is a thin coating of good quality wood used in furniture-making to hide lower quality wood underneath. Computer scientists use this metaphor to describe a small piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

## ***18.4 Pushing and popping***

A stack is a generic data structure, which means that we can add any type of item to it. The following example pushes two integers and a string onto the stack:

```
s2 = Stack2()
s2.Push(54)
s2.Push(45)
s2.Push("+")
```

We can use `IsEmpty` and `pop` to remove and print all of the items on the stack:

```
while not s2.IsEmpty():  
    print s2.Pop()
```

The output is + 45 54. In other words, we just used a stack to print the items backward! Granted, it's not the standard format for printing a list, but by using a stack, it was remarkably easy to do.

You should compare this bit of code to the implementation of `printBackward` in Section 17.4. There is a natural parallel between the recursive version of `printBackward` and the stack algorithm here. The difference is that `printBackward` uses the runtime stack to keep track of the nodes while it traverses the list, and then prints them on the way back from the recursion. The stack algorithm does the same thing, except that it uses a `Stack2` object instead of the runtime stack.

## ***18.5 Using a stack to evaluate postfix***

In most programming languages, mathematical expressions are written with the operator between the two operands, as in `1+2`. This format is called infix. An alternative used by some calculators is called postfix. In postfix, the operator follows the operands, as in `1 2 +`.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack:

1. Starting at the beginning of the expression, get one term (operator or operand) at a time.
  - a. If the term is an operand, push it on the stack.
  - b. If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.
2. When you get to the end of the expression, there should be exactly one operand left on the stack. That operand is the result.

*As an exercise, apply this algorithm to the expression  $1\ 2 + 3 *$ .*

This example demonstrates one of the advantages of postfix -- there is no need to use parentheses to control the order of operations. To get the same result in infix, we would have to write  $(1 + 2) * 3$ .

*As an exercise, write a postfix expression that is equivalent to  $1 + 2 * 3$ .*

## 18.6 Parsing

To implement the previous algorithm, we need to be able to traverse a string and break it into operands and operators. This process is an example of parsing, and the results -- the individual chunks of the string -- are called tokens. You might remember these words from Chapter 1.

Boo provides a `split` method in both the string and Regular Expressions modules. The function `string.Split` splits a string into a list using a single character as a delimiter. For example:

```
sample = "Now is the time"
for stringlet in sample.Split(char(' ')):
    print stringlet
#the above outputs: 'Now', 'is', 'the',
'time'
```

In this case, the delimiter is the space character, so the string is split at each space.

The function `Regex.Split` is more powerful, allowing us to provide a regular expression instead of a delimiter. A regular

expression is a way of specifying a set of strings. For example, [A-z] is the set of all letters and [0-9] is the set of all numbers. The ^ operator negates a set, so [^0-9] is the set of everything that is not a number, which is exactly the set we want to use to split up postfix expressions:

```
import System.Text.RegularExpressions
for stringlet in Regex.Split("123+456*/",
    "[^0-9]"):
    print stringlet
#the above outputs: '123', '+', '456', '*',
'', '/', ''
```

The resulting list includes the operands 123 and 456 and the operators \* and /. It also includes two empty strings that are inserted after the operands.

## **18.7 Evaluating postfix**

To evaluate a postfix expression, we will use the parser from the previous section and the algorithm from the section before that. To keep things simple, we'll start with an evaluator that only implements the operators + and \*:

```
import System.Text.RegularExpressions

def EvalPostfix(expr as string):
    tokenList = Regex.Split(expr, "[^0-9]")
    stack = Stack2()
    for token in tokenList:
        if token.ToString() == '' or token == ' ':
            continue
        elif token.ToString is null:
            continue
        elif token.ToString() == '+':
            sum =
```

```

System.Convert.ToInt16(stack.Pop()) +
System.Convert.ToInt16(stack.Pop())
    stack.Push(sum)
    elif token == '*':
        product =
System.Convert.ToInt16(stack.Pop()) *
System.Convert.ToInt16(stack.Pop())
        stack.Push(product)
    else:
        stack.Push(token)
return stack.Pop()

```

The first condition takes care of spaces and empty strings. The next two conditions handle operators. We assume, for now, that anything else must be an operand. Of course, it would be better to check for erroneous input and report an error message, but we'll get to that later.

Let's test it by evaluating the postfix form of  $(56+47)*2$ :

```

print EvalPostfix ("56 47 + 2 *")
#the above outputs: 206

```

## 18.8 Clients and providers

One of the fundamental goals of an ADT is to separate the interests of the provider, who writes the code that implements the ADT, and the client, who uses the ADT. The provider only has to worry about whether the implementation is correct -- in accord with the specification of the ADT -- and not how it will be used.

Conversely, the client assumes that the implementation of the ADT is correct and doesn't worry about the details. When you are using one of Boo's built-in types, you have the luxury of thinking exclusively as a client. Of course, when you implement an ADT, you also have to write client code to test it. In that case, you play



both roles, which can be confusing. You should make some effort to keep track of which role you are playing at any moment.

## **18.9 Glossary**

*abstract data type (ADT)*: A data type (usually a collection of objects) that is defined by a set of operations but that can be implemented in a variety of ways.

*interface*: The set of operations that define an ADT.

*implementation*: Code that satisfies the syntactic and semantic requirements of an interface.

*client*: A program (or the person who wrote it) that uses an ADT.

*provider*: The code (or the person who wrote it) that implements an ADT.

*veneer*: A class definition that implements an ADT with method definitions that are invocations of other methods, sometimes with simple transformations. The veneer does no significant work, but it improves or standardizes the interface seen by the client.

*generic data structure*: A kind of data structure that can contain data of any type.

*infix*: A way of writing mathematical expressions with the operators between the operands.

*postfix*: A way of writing mathematical expressions with the operators after the operands.

*parse*: To read a string of characters or tokens and analyze its grammatical structure.

*token*: A set of characters that are treated as a unit for purposes of parsing, such as the words in a natural language.

*delimiter*: A character that is used to separate tokens, such as punctuation in a natural language.

## **Chapter 19.0: Queues**

This chapter presents two ADTs: the Queue and the Priority Queue. In real life, a queue is a line of customers waiting for service of some kind. In most cases, the first customer in line is the next customer to be served. There are exceptions, though. At airports, customers whose flights are leaving soon are sometimes taken from the middle of the queue. At supermarkets, a polite customer might let someone with only a few items go first.

The rule that determines who goes next is called the queuing policy. The simplest queuing policy is called FIFO, for “first-in-first-out.” The most general queuing policy is priority queuing, in which each customer is assigned a priority and the customer with the highest priority goes first, regardless of the order of arrival. We say this is the most general policy because the priority can be based on anything: what time a flight leaves; how many groceries the customer has; or how important the customer is. Of course, not all queuing policies are “fair,” but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations. The difference is in the semantics of the operations: a queue uses the FIFO policy; and a priority queue (as the name suggests) uses the priority queuing policy.

### **19.1 The Queue ADT**

The Queue ADT is defined by the following operations:

*init*: Initialize a new empty queue.

*insert*: Add a new item to the end of the queue.

*remove*: Remove and return an item from the queue. The item that is returned is the first one that was added.

*isEmpty*: Check whether the queue is empty.

## 19.2 Linked Queue

The first implementation of the Queue ADT we will look at is called a linked queue because it is made up of linked Node objects. Here is the class definition:

```
class Node:
    [property(Cargo)] _cargo as Object
    [property(Next)] _next as Node

    def constructor(cargo as object):
        Cargo = cargo

class Queue:
    [property (Length)] _length as double = 0
    [property (Head)] _head as Node

    def IsEmpty():
        return (Length == 0)
    def Insert(cargo):
        node = Node(cargo)
        if Head == null:
            # if list empty the new node goes 1st
            Head = node
        else:
            # find the last node in the list
            last = Head
            while last.Next is not null:
                last = last.Next
            # append the new node
            last.Next = node
        Length = Length + 1
    def Remove():
        if Length == 0:
            return
```

```
cargo = Head.Cargo
Head = Head.Next
Length = Length - 1
return cargo
```

The methods `isEmpty` and `remove` are identical to the `LinkedList` methods `isEmpty` and `removeFirst`. The `insert` method is new and a bit more complicated. We want to insert new items at the end of the list. If the queue is empty, we just set `head` to refer to the new node.

Otherwise, we traverse the list to the last node and tack the new node on the end. We can identify the last node because its `next` attribute is `null`. There are two invariants for a properly formed `Queue` object. The value of `length` should be the number of nodes in the queue, and the last node should have `next` equal to `null`. Convince yourself that this method preserves both invariants.

As an exercise, enhance the `remove` method to handle a case where there is nothing to remove.

### ***19.3 Performance characteristics***

Normally when we invoke a method, we are not concerned with the details of its implementation. But there is one “detail” -- we might want to know the performance characteristics of the method. How long does it take, and how does the run time change as the number of items in the collection increases?

First, look at `remove`. There are no loops or function calls here, suggesting that the run time of this method is the same every time. Such a method is called a constant-time operation. In reality, the method might be slightly faster when the queue is empty since it skips the body of the conditional, but that difference is not significant.

The performance of insert is very different. In the general case, we have to traverse the queue to find the last element. This traversal takes time proportional to the length of the queue. Since the run time is a linear function of the length, this method is called linear time. Compared to constant time, that's very bad.

## ***19.4 Improved Linked Queue***

We would like an implementation of the Queue ADT that can perform all operations in constant time. One way to do that is to modify the Queue class so that it maintains a reference to both the first and the last node, as shown in the figure:

The ImprovedQueue implementation looks like this:

```
class ImprovedQueue:
    [property(Length)] _length as double = 0
    [property(Head)] _head as Node
    [property>Last)] _last as Node

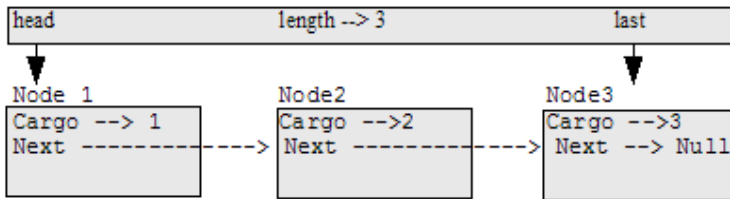
    def IsEmpty():
        return (Length == 0)
```

So far, the only change is the attribute last. It is used in insert and remove methods:

```
class ImprovedQueue:
    ...
    def Insert(cargo):
        node = Node(cargo)
        node.Next = null
        if Length == 0:
            # if list is empty, the new node is
            head and last
            Head = node
```

```
else:
    # find the last node
    last = Last
    # append the new node
    last.Next = node
Last = node
Length = Length + 1
```

Since last keeps track of the last node, we don't have to search for it. As a result, this method is constant time. There is a price to pay for that speed. We have to add a special case to `remove` to set `last` to null when the last node is removed:



```
class ImprovedQueue:
...
    def Remove():
        if Length == 0:
            return
        cargo = Head.Cargo
        Head = Head.Next
        Length = Length - 1
        if Length == 0:
            Last = null
        return cargo
```

This implementation is more complicated than the Linked Queue implementation, and it is more difficult to demonstrate that it is correct. The advantage is that we have achieved the goal -- both

insert and remove are constant-time operations.

As an exercise, write an implementation of the Queue ADT using a list. Compare the performance of this implementation to the `ImprovedQueue` for a range of queue lengths.

## **19.5 Priority queue**

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics. Again, the interface is:

`init` : Initialize a new empty queue.

`insert`: Add a new item to the queue.

`remove`: Remove and return an item from the queue. The item that is returned is the one with the highest priority.

`isEmpty`: Check whether the queue is empty.

The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is the item in the queue that has the highest priority. What the priorities are and how they compare to each other are not specified by the Priority Queue implementation. It depends on which items are in the queue.

For example, if the items in the queue have names, we might choose them in alphabetical order. If they are bowling scores, we might go from highest to lowest, but if they are golf scores, we would go from lowest to highest. As long as we can compare the items in the queue, we can find and remove the one with the highest priority.

This implementation of `PriorityQueue` has as an attribute a list that contains the items in the queue.



```
class PriorityQueue:
    [property(Items)] _items = []

    def IsEmpty():
        return Items == []
    def Insert(item):
        Items.Add(item)
```

The initialization method, `isEmpty`, and `insert` are all veneers on list operations. The only interesting method is `remove`:

```
class PriorityQueue:
...
    def Remove():
        maxi as double = 0
        for i in range(1,len(Items)):
            #Code would have to be added to
            #override the > operator for this
            #procedure to work for other than
            #just numerics
            if (Items[i] as duck) > (Items[maxi]
as duck):
                maxi = i
            item = Items[maxi]
            Items.RemoveAt(maxi)
            return item
```

At the beginning of each iteration, `maxi` holds the index of the biggest item (highest priority) we have seen so far. Each time through the loop, the program compares the `i`-eth item to the champion. If the new item is bigger, the value of `maxi` is set to `i`.

When the for statement completes, `maxi` is the index of the

biggest item. This item is removed from the list and returned.  
Let's test the implementation:

```
q = PriorityQueue()
q.Insert(11)
q.Insert(17)
q.Insert(14)
q.Insert(13)
while not q.IsEmpty():
    print q.Remove() #outputs: 17 14 13 11
```

If the queue contains simple numbers, they are removed in numerical order, from highest to lowest.

## 19.6 The Golfer Class

As an example of an object with an unusual definition of priority, let's implement a class called *Golfer* that keeps track of the names and scores of golfers.

```
class Golfer:
    [property(Name)] _name as string
    [property(Score)] _score as double

    override def ToString():
        return "$Name, $Score"
        #alternate syntax: return "%-16s: %d" %
(Name, Score)
    static def op_LessThan(x as Golfer, y as
Golfer):
        if x.Score < y.Score:
            return 1
        else:
            return 0
    static def op_GreaterThan(x as Golfer, y
as Golfer):
```

```
#reverse the logic. Return false if
greater than so lowest score is 'higher'
    if x.Score > y.Score:
        return 0
    else:
        return 1
def constructor(name as string, score as
double):
    Name = name
    Score= score
```

ToString outputs the names and score. ToString is called by the print macro. By overriding the ToString whenever we use “print Golfer” the print macro returns the name and score of the golfer instead of the class. Next we override the greater than operator so the lowest score gets highest priority. The greater than operator is called op\_GreaterThan. Overloaded operators must always be defined as static. The less than operator is also defined, but this is done only as a matter of practice and not utilized by the program.

Now we are ready to test the priority queue with the Golfer class:

```
match = PriorityQueue()
match.Insert(Golfer("Tiger Woods", 61))
match.Insert(Golfer("Hal Sutton", 69))
match.Insert(Golfer("Phil Mickelson", 72))
while (match.IsEmpty() == false):
    print match.Remove( )
```

As an exercise, write an implementation of the Priority Queue ADT using a linked list. You should keep the list sorted so that removal is a constant time operation. Compare the performance of this implementation with the boo list implementation.

## 19.7 Glossary

*queue*: An ordered set of objects waiting for a service of some kind.

*Queue*: An ADT that performs the operations one might perform on a queue.

*queuing policy*: The rules that determine which member of a queue is removed next.

*FIFO*: "First In, First Out," a queuing policy in which the first member to arrive is the first to be removed.

*priority queue*: A queueing policy in which each member has a priority determined by external factors. The member with the highest priority is the first to be removed.

*Priority Queue*: An ADT that defines the operations one might perform on a priority queue.

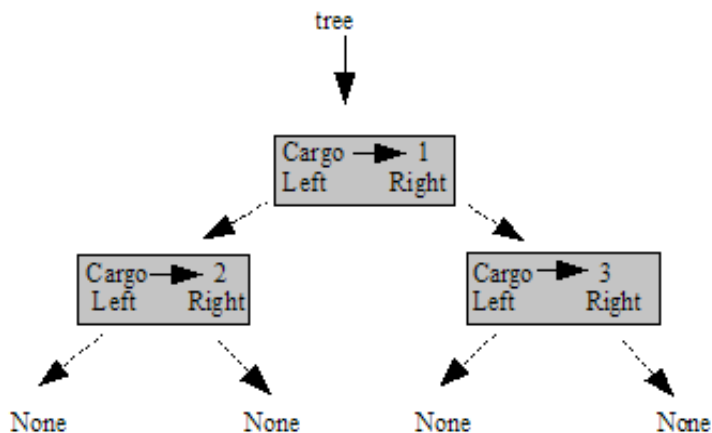
*linked queue*: An implementation of a queue using a linked list.

*constant time*: An operation whose run time does not depend on the size of the data structure.

*linear time*: An operation whose run time is a linear function of the size of the data structure.

## Chapter 20.0: Trees

Like linked lists, trees are made up of nodes. A common kind of tree is a binary tree, in which each node contains a reference to two other nodes (possibly null). These references are referred to as the left and right subtrees. Like list nodes, tree nodes also contain cargo. A state diagram for a tree looks like this:



To avoid cluttering up the picture, we often omit the Nodes. The top of the tree (the node the tree refers to) is called the root. In keeping with the tree metaphor, the other nodes are called branches and the nodes at the tips with null references are called leaves. It may seem odd that we draw the picture with the root at the top and the leaves at the bottom, but that is not the strangest thing.

To make things worse, computer scientists mix in another metaphor -- the family tree. The top node is sometimes called a parent and the nodes it refers to are its children. Nodes with the same parent are called siblings.

Finally, there is a geometric vocabulary for talking about trees. We already mentioned left and right, but there is also “up” (toward the parent/root) and “down” (toward the children/leaves). Also, all of the nodes that are the same distance from the root comprise a level of the tree. We probably don't need three metaphors for talking about trees, but there they are.

Like linked lists, trees are recursive data structures because they are defined recursively. A tree is either:

- the empty tree, represented by None, or
- a node that contains an object reference and two tree references.

## ***20.1 Building trees***

The process of assembling a tree is similar to the process of assembling a linked list. Each constructor invocation builds a single node.

```
class Tree:
    [property(Cargo)] _cargo as object
    [property(Left)] _left as Tree
    [property(Right)] _right as Tree
    def constructor(cargo):
        Cargo = cargo
    def constructor(cargo, left, right):
        Cargo = cargo
        Left = left
        Right = right
    def ToString():
        return Cargo
```

The cargo can be any type, but the left and right parameters should be tree nodes. left and right are optional; the default value

is null. To print a node, we just print the cargo. One way to build a tree is from the bottom up. Allocate the child nodes first:

```
left = Tree(2)
right = Tree(3)
```

Then create the parent node and link it to the children:

```
tree = Tree(1, left, right)
```

We can write this code more concisely by nesting constructor invocations:

```
tree = Tree(1, Tree(2), Tree(3))
```

Either way, the result is the tree at the beginning of the chapter.

## **20.2 Traversing trees**

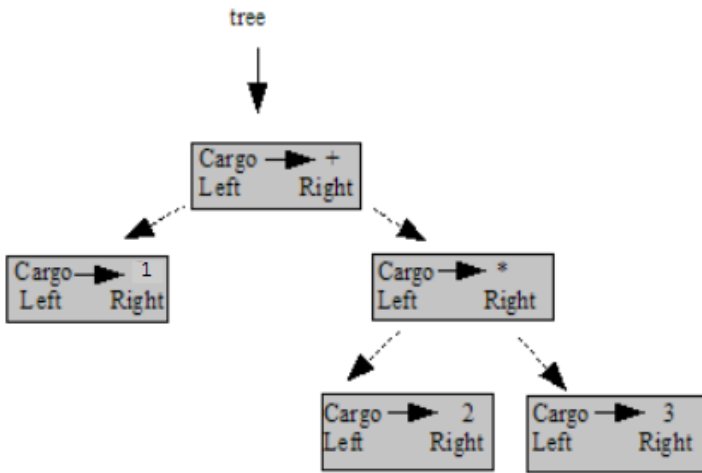
Any time you see a new data structure, your first question should be, “How do I traverse it?” The most natural way to traverse a tree is recursively. For example, if the tree contains integers as cargo, this function returns their sum:

```
def total(tree as Tree) as int:
    if tree == null:
        return 0
    return total(tree.Left) +
total(tree.Right) +
Convert.ToInt32(tree.Cargo)
```

The base case is the empty tree, which contains no cargo, so the sum is 0. The recursive step makes two recursive calls to find the sum of the child trees. When the recursive calls complete, we add the cargo of the parent and return the total.

## 20.3 Expression trees

A tree is a natural way to represent the structure of an expression. Unlike other notations, it can represent the computation unambiguously. For example, the infix expression  $1 + 2 * 3$  is ambiguous unless we know that the multiplication happens before the addition. This expression tree represents the same computation:



The nodes of an expression tree can be operands like 1 and 2 or operators like + and \*. Operands are leaf nodes; operator nodes contain references to their operands. (All of these operators are binary, meaning they have exactly two operands.) We can build this tree like this:

```
tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
```

Looking at the figure, there is no question what the order of operations is; the multiplication happens first in order to compute



the second operand of the addition.

Expression trees have many uses. The example in this chapter uses trees to translate expressions to postfix, prefix, and infix. Similar trees are used inside compilers to parse, optimize, and translate programs.

## **20.4 Tree traversal**

We can traverse an expression tree and print the contents like this:

```
def printTree(tree as Tree):
    if tree == null:
        return
    print tree.Cargo
    printTree(tree.Left)
    printTree(tree.Right)
```

In other words, to print a tree, first print the contents of the root, then print the entire left subtree, and then print the entire right subtree. This way of traversing a tree is called a pre-order, because the contents of the root appear before the contents of the children. For the previous example, the output is:

```
treeTest = Tree('+', Tree(1), Tree('*',
Tree(2), Tree(3)))
printTree(treeTest) #outputs: + 1 + 2 3
```

This format is different from both postfix and infix; it is another notation called prefix, in which the operators appear before their operands. You might suspect that if you traverse the tree in a different order, you will get the expression in a different notation. For example, if you print the subtrees first and then the root node, you get:

```
def printTreePostorder(tree as Tree):
    if tree == null:
```

```

    return
    printTreePostorder(tree.Left)
    printTreePostorder(tree.Right)
    print tree.Cargo

```

The result, 1 2 3 \* +, is in postfix! This order of traversal is called postorder. Finally, to traverse a tree in order, you print the left tree, then the root, and then the right tree:

```

def printTreeInOrder(tree as Tree):
    if tree == null:
        return
    printTreeInOrder(tree.Left)
    print tree.Cargo
    printTreeInOrder(tree.Right)

```

The result is 1 + 2 \* 3, which is the expression in infix. To be fair, we should point out that we have omitted an important complication. Sometimes when we write an expression in infix, we have to use parentheses to preserve the order of operations. So an in order traversal is not quite sufficient to generate an infix expression. Nevertheless, with a few improvements, the expression tree and the three recursive traversals provide a general way to translate expressions from one format to another.

As an exercise, modify printTreeInOrder so that it puts parentheses around every operator and pair of operands. Is the output correct and unambiguous? Are the parentheses always necessary? If we do an in order traversal and keep track of what level in the tree we are on, we can generate a graphical representation of a tree:

```

def printTreeIndented(tree as Tree, level as int):
    if tree == null:
        return
    printTreeIndented(tree.Right, level+1)
    print ' '*level +

```

```
Convert.ToString(tree.Cargo)
  printTreeIndented(tree.Left, level+1)
```

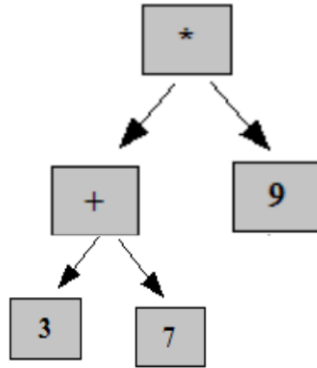
The parameter `level` keeps track of where we are in the tree. Each time we make a recursive call, we pass `level+1` because the child's level is always one greater than the parent's. Each item is indented by two spaces per level. The result for the example tree is:

```
>>> printTreeIndented(treeTest, 0)
      3
    *
      2
+
      1
```

If you look at the output sideways, you see a simplified version of the original figure.

## ***20.5 Building an expression tree***

In this section, we parse infix expressions and build the corresponding expression trees. For example, the expression  $(3+7)*9$  yields the following tree



:

Notice that we have simplified the diagram by leaving out the names. The parser we will write handles expressions that include numbers, parentheses, and the operators + and \*. We assume that the input string has already been tokenized into a boo list. The token list for (3+7)\*9 is:

```
[ '(', 3, '+', 7, ')', '*', 9, 'end' ]
```

The end token is useful for preventing the parser from reading past the end of the list. As an exercise, let's write a function that takes an expression string and returns a token list. The first function we'll write is `getToken`, which takes a token list and an expected token as parameters. It compares the expected token to the first token on the list: if they match, it removes the token from the list and returns true; otherwise, it returns false:

```
def getToken(tokenList as List, expected as
string):
    if tokenList[0] == expected:
        tokenList.Pop(0)
        return 1
    else:
        return 0
```

Since `tokenList` refers to a mutable object, the changes made here are visible to any other variable that refers to the same object.

The next function, `getNumber`, handles operands. If the next token in `tokenList` is a number, `getNumber` removes it and returns a leaf node containing the number; otherwise, it returns `None`.

```
def getNumber(tokenList as List) as Tree:
    if tokenList[0].GetType() ==
typeof(double):
        return null
    x = tokenList.Pop(0)
    return Tree (x, null, null)
```

Before continuing, we should test `getNumber` in isolation. We assign a list of numbers to `tokenList`, extract the first, print the result, and print what remains of the token list:

```
tokenList = [9, 11, 'end']
x = getNumber(tokenList)
printTreePostorder(x)      #outputs: 9
print tokenList             #outputs: [11, 'end']
```

The next method we need is `getProduct`, which builds an expression tree for products. A simple product has two numbers as operands, like  $3 * 7$ . Here is a version of `getProduct` that handles simple products.

```
def getProduct(tokenList as List) as Tree:
    a = getNumber(tokenList)
    if getToken(tokenList, "*"):
        b = getNumber(tokenList)
        return Tree('*', a, b)
    else:
        return a
```

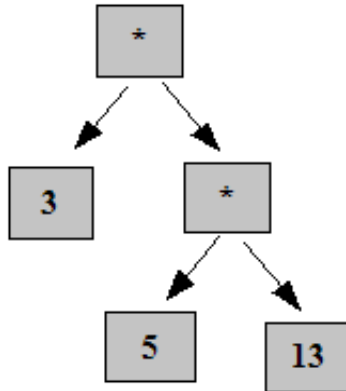
Assuming that `getNumber` succeeds and returns a singleton tree, we assign the first operand to `a`. If the next character is `*`, we get the second number and build an expression tree with `a`, `b`, and the operator. If the next character is anything else, then we just return the leaf node with `a`. Here are two examples:

```
tokenList = [9, '*', 11, 'end']
tree = getProduct(tokenList)
printTreePostorder(tree) #outputs: 9 11 *

tokenList = [9, '+', 11, 'end']
tree = getProduct(tokenList)
printTreePostorder(tree) #outputs: 9
```

The second example implies that we consider a single operand to be a kind of product. This definition of “product” is counter intuitive, but it turns out to be useful. Now we have to deal with compound products, like like  $3 * 5 * 13$ . We treat this expression as a product of products, namely  $3 * (5 * 13)$ .

The resulting tree is:



With a small change in `getProduct`, we can handle an arbitrarily long product:

```
def getProduct(tokenList) as Tree:
    a = getNumber(tokenList)
    if getToken(tokenList, '*'):
        b = getProduct(tokenList) # this line
changed
        return Tree('*', a, b)
    else:
        return a
```

In other words, a product can be either a singleton or a tree with `*` at the root, a number on the left, and a product on the right. This kind of recursive definition should be starting to feel familiar. Let's test the new version with a compound product:

```
tokenList = [2, '*', 3, '*', 5, '*', 7,
'end']
```

```
tree = getProduct(tokenList)
printTreePostorder(tree) #outputs: 2 3 5 7 *
* *
```

Next we will add the ability to parse sums. Again, we use a slightly counter intuitive definition of “sum.” For us, a sum can be a tree with + at the root, a product on the left, and a sum on the right. Or, a sum can be just a product.

If you are willing to play along with this definition, it has a nice property: we can represent any expression (without parentheses) as a sum of products. This property is the basis of our parsing algorithm. `getSum` tries to build a tree with a product on the left and a sum on the right. But if it doesn't find a +, it just builds a product.

```
def getSum(tokenList as List) as Tree:
    a = getProduct(tokenList)
    if getToken(tokenList, "+"):
        b = getSum(tokenList)
        return Tree ("+", a, b)
    else:
        return a
```

Let's test it with  $9 * 11 + 5 * 7$ :

```
tokenList = [9, '*', 11, '+', 5, '*', 7,
'end']
tree = getSum(tokenList)
printTreePostorder(tree) #output : 9 11 * 5
7 * +
```

We are almost done, but we still have to handle parentheses. Anywhere in an expression where there can be a number, there can also be an entire sum enclosed in parentheses. We just need to modify `getNumber` to handle sub-expressions:



```
def getNumber(tokenList as List) as Tree:
    if getToken(tokenList, "("):
        y = getSum(tokenList)
        getToken(tokenList, ")")
        return y
    else:
        if tokenList[0].GetType() ==
typeof(double):
            return null
        x = tokenList.Pop(0)
        return Tree (x, null, null)
```

Let's test this code with  $9 * (11 + 5) * 7$ :

```
tokenList = [9, '*', '(', 11, '+', 5, ')',
              '*', 7, 'end']
tree = getSum(tokenList)
printTreePostorder(tree) #outputs: 9 11 5 +
7 * *
```

The parser handled the parentheses correctly; the addition happens before the multiplication. In the final version of the program, it would be a good idea to give `getNumber` a name more descriptive of its new role. Throughout the parser, we've been assuming that expressions are well-formed. For example, when we reach the end of a sub-expression, we assume that the next character is a close parenthesis. If there is an error and the next character is something else, we should deal with it.

```
def getNumber(tokenList as List) as Tree:
    if getToken(tokenList, '('):
        x = getSum(tokenList)
    if not getToken(tokenList, ')'):
        raise Exception('missing parenthesis')
    return x
else:
```

```
# the rest of the function omitted
```

The `raise` statement creates an exception; in this case we throw a message that reads 'missing parenthesis'. If the function that called `getNumber`, or one of the other functions in the traceback, handles the exception, then the program can continue. Otherwise, `boo` will print an error message and quit.

As an exercise, find other places in these functions where errors can occur and add appropriate `raise` statements. Test your code with improperly formed expressions.

## 20.6 The animal tree

In this section, we develop a small program that uses a tree to represent a knowledge base. The program interacts with the user to create a tree of questions and animal names. Here is a sample run:

*Are you thinking of an animal?* yes

*Is it a bird?* no

*What is the animals name?* dog

*What question would distinguish a dog from a bird?* Can it fly?

*If the animal were dog the answer would be?* no

*Are you thinking of an animal?* yes

*Can it fly?* no

*Is it a dog?* no

*What is the animals name?* cat

*What question would distinguish a cat from a dog?* Does it bark?

*If the animal were cat the answer would be?* no

*Are you thinking of an animal?* yes

*Can it fly?* no

*Learning to Program with Boo*

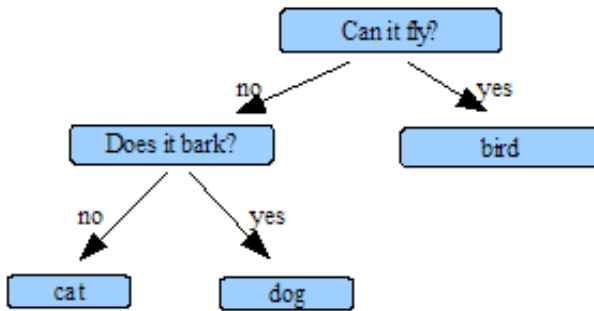
*Does it bark? yes*

*Is it a dog? yes*

*I rule!*

*Are you thinking of an animal? no*

Here is the tree this dialog builds:



At the beginning of each round, the program starts at the top of the tree and asks the first question. Depending on the answer, it moves to the left or right child and continues until it gets to a leaf node. At that point, it makes a guess.

If the guess is not correct, it asks the user for the name of the new animal and a question that distinguishes the (bad) guess from the new animal. Then it adds a node to the tree with the new question and the new animal.

Here is the code:

```
class Tree:
    [property(Cargo)] _cargo as object
    [property(Left)] _left as Tree
    [property(Right)] _right as Tree
```

```

def constructor(cargo):
    Cargo = cargo
def constructor(cargo, left, right):
    Cargo = cargo
    Left = left
    Right = right
def ToString():
    return Cargo

def yes(ques as string):
    #from string import lower
    #ans = lower(raw_input(ques))
    #return (ans[0] == 'y')
    ques = ques.ToLower()
    return (ques[0] == char('y'))

def animal():
    # start with a singleton
    lvl2b = Tree("Dog")
    lvl2a = Tree("Cat")
    lvl1b = Tree("Bird")
    lvl1a = Tree("Does it Bark ", lvl2a, lvl2b)
    root = Tree("Can it fly ", lvl1a, lvl1b)

    # loop until the user quits
    # while 1:
    Console.WriteLine('Are you thinking of an
animal?')
    question as string = Console.ReadLine()
    if yes(question):
        # walk the tree
        tree = root
        while tree.Left != null:
            #prompt = tree.getCargo() + "? "
            text as string = tree.Cargo.ToString()
+   "? "
            Console.WriteLine(text)

```

```
text = Console.ReadLine()
if yes(text):
    tree = tree.Right
    # make a guess
    guess as string =
tree.Cargo.ToString()
text = "Is it a " + guess + "? "
Console.WriteLine(text)
text = Console.ReadLine()
if yes(text):
    print "I rule!"
    return
else:
    print "Come and Play again"
    return
else:
    tree = tree.Left
    #continue
    Console.WriteLine("Is it a " +
tree.Cargo.ToString() + "?")
    else:
        print "Come and Play again"
animal()
print "Press Enter to continue . . . "
Console.ReadLine()
```

The function `yes` is a helper; it prints a prompt and then takes input from the user. If the response begins with `y` or `Y`, the function returns `true`:

```
def yes(ques as string):
    #from string import lower
    #ans = lower(raw_input(ques))
    #return (ans[0] == 'y')
    ques = ques.ToLower()
```

```
return (ques[0] == char('y'))
```

The `while` loop walks the tree from top to bottom, guided by the user's responses.

## 20.7 Glossary

*binary tree*: A tree in which each node refers to zero, one, or two dependent nodes.

*root*: The topmost node in a tree, with no parent.

*leaf*: A bottom-most node in a tree, with no children.

*parent*: The node that refers to a given node.

*child*: One of the nodes referred to by a node.

*siblings*: Nodes that share a common parent.

*level*: The set of nodes equidistant from the root.

*binary operator*: An operator that takes two operands.

*subexpression*: An expression in parentheses that acts as a single operand in a larger expression.

*pre-order*: A way to traverse a tree, visiting each node before its children.

*prefix notation*: A way of writing a mathematical expression with each operator appearing before its operands.

*postorder*: A way to traverse a tree, visiting the children of each node before the node itself.

*in-order*: A way to traverse a tree, visiting the left subtree, then the root, and then the right subtree.

## Appendix A: Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

- \* Syntax errors are produced by boo when it is translating the source code into byte code. They usually indicate that there is something wrong with the way the program is formed.
- \* Runtime errors are produced by the runtime system if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes a runtime error of “maximum recursion depth exceeded.”
- \* Semantic errors are problems with a program that compiles and runs but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an unexpected result. The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

### ***A.1 Syntax errors***

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where boo noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line. If you are building the program incrementally, you should have a good idea about where the error



is. It will be in the last line you added. If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

More sophisticated text editors or an Integrated Development Environment (IDE) greatly reduce the effort to resolve many common programming errors. SharpDevelop

( <http://www.icsharpcode.net/OpenSource/SD/Default.aspx> ) is a fine, free example of an IDE.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a keyword for a variable name.
2. Check that you have a colon at the end of the header of every compound statement, including for, while, if, class, and def statements.
3. Check that indentation is consistent. You may indent with either spaces or tabs but it's dangerous to intermix them. Each level should be nested the same amount.
4. Make sure that any strings in the code have matching quotation marks.
5. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
6. An unclosed bracket (, {, or [ makes boo continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.

7. Check for the classic `=` instead of `==` inside a conditional.

If nothing works, move on to the next section...

### **A.1.1 I can't get my program to run no matter what I do.**

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one boo is trying to run. If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run (or import) it again. If the compiler doesn't find the new error, there is probably something wrong with the way your environment is set up.

If this happens, one approach is to start again with a new program like "Hello, World!," and make sure you can get a known program to run. Then gradually add the pieces of the new program to the working one.

## ***A.2 Runtime errors***

Once your program is syntactically correct, boo can import it and at least start running it. What could possibly go wrong?

### **A.2.1 My program does absolutely nothing.**

This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the "Flow of Execution" section below.

## **A.2.2 My program hangs.**

If a program stops and seems to be doing nothing, we say it is “hanging.” Often that means that it is caught in an infinite loop or an infinite recursion.

\* If there is a particular loop that you suspect is the problem, add a print statement immediately before the loop that says “entering the loop” and another immediately after that says “exiting the loop.” Run the program. If you get the first message and not the second, you’ve got an infinite loop. Go to the “Infinite Loop” section below.

\* Most of the time, an infinite recursion will cause the program to run for a while and then produce a “RuntimeError: Maximum recursion depth exceeded” error. If that happens, go to the “Infinite Recursion” section below.

If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the “Infinite Recursion” section.

\* If neither of those steps works, start testing other loops and other recursive functions and methods.

\* If that doesn’t work, then it is possible that you don’t understand the flow of execution in your program. Go to the “Flow of Execution” section below.

### **Infinite Loop**

If you think you have an infinite loop and you think you know what loop is causing the problem, add a print statement at the end of the loop that prints the values of the variables in the condition and the value of the condition. For example:

```
while x > 0 and y < 0 :  
    # do something to x  
    # do something to y  
    print "x: ", x
```

```
print "y: ", y
print "condition: ", (x > 0 and y < 0)
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be false. If the loop keeps going, you will be able to see the values of *x* and *y*, and you might figure out why they are not being updated correctly.

## Infinite Recursion

Most of the time, an infinite recursion will cause the program to run for a while and then produce a Maximum recursion depth exceeded error. If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the function or method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a print statement at the beginning of the function or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

## Flow of Execution

If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each function with a message like “entering function foo”.

Now when you run the program, it will print a trace of each function as it is invoked.

### **A.2.3 When I run the program I get an exception.**

If something goes wrong during runtime, Boo prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.

The traceback identifies the function that is currently running, and then the function that invoked it, and then the function that invoked that, and so on. In other words, it traces the path of function invocations that got you to where you are. It also includes the line number in your file where each of these calls occurs.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

`DivideByZero`:

\* The values passed in to your variables are such that boo has a zero in the denominator. Consider reformulating or add exception handling.

`IndexOutOfRangeException`: The index you are using to access a string or array that is greater than its length minus one. Immediately before the site of the error, add a print statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

### **A.2.4 I added so many print statements I get inundated with output.**

One of the problems with using print statements for debugging is

that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out print statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are sorting an array, sort a small array. If the program takes input from the user, give it the simplest input that causes the problem.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smaller functions and testing them separately.

Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on. Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

### ***A.3 Semantic errors***

In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast. You will often wish that you could slow the program down to human speed, and with

some debuggers you can. But the time it takes to insert a few well-placed print statements is often short compared to setting up the debugger, inserting and removing breakpoints, and “walking” the program to where the error is occurring.<sup>13</sup>

### **A.3.1 My program doesn't work.**

You should ask yourself these questions:

- \* Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
- \* Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
- \* Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to functions or methods in other boo modules. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

---

<sup>13</sup> SharpDevelop allows the setting of breakpoints, and single stepping thru a program. It also allows you to watch variables and the call stack.

### A.3.2 I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
hands[i].addCard
(hands[findNeighbor(i)].popCard())
```

This can be rewritten as:

```
neighbor = findNeighbor (i)
pickedCard = hands[neighbor].popCard()
hands[i].addCard (pickedCard)
```

The expanded version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression  $x / 2^{1/4}$  into boo, you might write:

```
y = x / 2 * 1 / 4
```

That is not correct because multiplication and division have the same precedence and are evaluated from left to right.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
y = x / (2 * 1/4)
```

Whenever you are not sure of the order of evaluation, use



parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the rules of precedence.

### **A.3.3 I've got a function or method that doesn't return what I expect.**

If you have a return statement with a complex expression, you don't have a chance to print the return value before returning. Again, you can use a temporary variable. For example, instead of: `return Hands[i].RemoveMatches()` you could write:

```
count = Hands[i].RemoveMatches()  
return count
```

Now you have the opportunity to display the value of `count` before returning.

### **A.3.4 I'm really, really stuck and I need help.**

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these effects:

- \* Frustration and/or rage.
- \* Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backward").
- \* Random-walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next? Try to explain what is or is not happening to someone (a sounding board).

Sometimes it just takes time to find a bug. We often find bugs when we are away from the computer and let our minds wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

### **A.3.5 No, I really need help.**

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing. Before you bring someone else in, make sure you have exhausted the techniques described here. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need:

- \* If there is an error message, what is it and what part of the program does it indicate?
- \* What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- \*What have you tried so far, and what have you learned?
- \*What version of the compiler are you using?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly. Remember, the goal is not just to make the program work. The goal is to learn how to make the program work.

## **Appendix B: GNU Free Documentation License**

## GNU Free Documentation

License

Version 1.1, March 2000

Copyright flc 2000 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### Preamble

The purpose of this License is to make a manual, textbook, or other written document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft,” which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

### ***B.1 Applicability and Definitions***

This License applies to any manual or other work that contains a

notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document," below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical, or political position regarding them. The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy

made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

## ***B.2 Verbatim Copying***

Examples of suitable formats for Transparent copies include plain ASCII without markup, Textinfo input format, LATEX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in Section 3. You may also lend copies, under the same conditions stated above, and you may publicly display copies.

### **B.3 Copying in Quantity**

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of

copies, to give them a chance to provide you with an updated version of the Document.

## ***B.4 Modifications***

You may copy and distribute a Modified Version of the Document under the conditions of Sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- \* Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- \* List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- \* State on the Title page the name of the publisher of the Modified Version, as the publisher.
- \* Preserve all the copyright notices of the Document.
- \* Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- \* Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- \* Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- \* Include an unaltered copy of this License.
- \* Preserve the section entitled "History," and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no



section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- \* Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- \* In any section entitled "Acknowledgments" or "Dedications," preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

- \* Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- \* Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.

- \* Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles. You may add a section entitled "Endorsements," provided it contains nothing but endorsements of your Modified Version by various parties -- for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity.

If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## ***B.5 Combining Documents***

You may combine the Document with other documents released under this License, under the terms defined in Section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work. In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements,” and any sections entitled “Dedications.”

You must delete all sections entitled “Endorsements.”

## ***B.6 Collections of Documents***

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## ***B.7 Aggregation with Independent Works***

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate,” and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document. If the Cover Text requirement of Section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

## ***B.8 Translation***

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of Section

4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

## ***B.9 Termination***

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense, or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## ***B.10 Future Revisions of This License***

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>. Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

## ***B.11 Addendum: How to Use This License for Your Documents***

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:  
Copyright f1c YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License."

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.