How to think like a computer scientist

Allen B. Downey and Jeffrey Elkner

Illustrations by John Dewey

How to think like a computer scientist

Python Version, First Edition

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The GNU Free Documentation License is available from www.gnu.org or by writing to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

The LaTeX source for this book, and more information about the Open Source Textbook project, is available from

`http://rocky.wellesley.edu/downey/ost`

and

`http://www.ibiblio.org/obp`

or by writing to Allen B. Downey, Computer Science Dept, Wellesley College, Wellesley, MA 02482.

This book was typeset by the authors using LaTeX and LyX, which are both free, open-source programs.

Illustrations by John Dewey

Contributor List

by Jeffrey Elkner

Perhaps the most exciting thing about a free content textbook is the possibility it creates for those using the book to collaborate in its development. I have been delighted by the many responses, suggestions, corrections, and words of encouragement I have received from people who have found this book to be useful, and who have taken the time to let me know about it.

Unfortunately, as a busy high school teacher who is working on this project in my spare time (what little there is of it ;-), I have been neglectful in giving credit to those who have helped with the book. I always planned to add an "Acknowlegdements" sections upon completion of the first stable version of the book, but as time went on it became increasingly difficult to even track those who had contributed.

Upon seeing the most recent version of Tony Kuphaldt's wonderful free text, "Lessons in Electric Circuits", I got the idea from him to create an ongoing "Contributor List" page which could be easily modified to include contributors as they come in.

My only regret is that many earlier contributors might be left out. I will begin as soon as possible to go back through old emails to search out the many wonderful folks who have helped me in this endeavour. In the mean time, if you find yourself missing from this list, please except my humble apologies and drop me an email letting me know about my oversight.

And so, without further delay, here is a listing of the contributors:

Fred Bremmer Fred submitted a correction in section 2.1. He can be reached at: `Fred.Bremmer@ubc.cu`

Jonah Cohen Jonah wrote the Perl scripts to convert the LaTeX source for this book into beautiful html. His web page is `jonah.ticalc.org` and his email is `JonahCohen@aol.com`

Courtney Gleason Courtney and Katherine Smith created the first version of `horsebet.py`, which is used as the case study for the last chapters of the book. Courtney can be reached at: `orion1558@aol.com`

Lee Harr Lee submitted corrections for sections 10.1 and 11.5. He can be reached at: `missive@linuxfreemail.com`

James Kaylin James is a student using the text. He has submitted numerous corrections. James can be reached by email at: `Jamarf@aol.com`

Eddie Lam Eddie has sent in numerous corrections to chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run. Eddie can be reached at: `epl@cs.monash.edu.au`

Man-Yong Lee Man-Yong sent in a correction to the example code in section 2.4. He can be reaced at: `yong@linuxkorea.co.kr`

Kevin Parks Kevin sent in valuable comments and suggestions as to how to improve the distribution of the book. He can be reached at: `cpsoct@lycos.com`

Katherine Smith Katherine and Courtney Gleason created the first version of `horsebet.py`, which is used as the case study for the last chapters of the book. Katherine can be reached at: `kss_0326@yahoo.com`

Craig T. Snydal Craig is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections, and can be reached at: `csnydal@drew.edu`

Ian Thomas Ian and his students are using the text in a programming course. They are the first ones to test out the chapters in the latter half of the book, and they have make numerous corrections and suggestions. Ian can be reached at: `ithomas@sd70.bc.ca`

Keith Verheyden Keith made correction in Section 3.11 and can be reached at: `kverheyd@glam.ac.uk`

Moshe Zadka Moshe has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book. He can be reached at: `moshez@math.huji.ac.il`

Contents

Chapter 1

The way of the program

The goal of this book, and this class, 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 tradeoffs 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, which is 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 Python programming language

The programming language you will be learning is Python. Python is an example of a **high-level language**; other high-level languages you might have heard of are C, C++, Perl and Java.

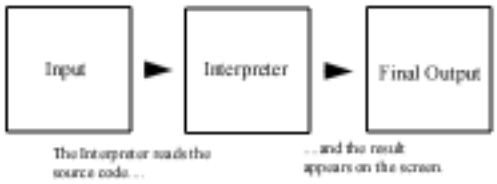
As you might infer from the name "high-level language," there are also **low-level languages**, sometimes referred to as machine language or assembly language. Loosely-speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be translated before they can run. This translation 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

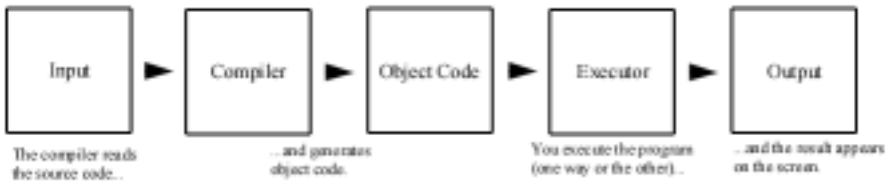
correct. Secondly, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are only used for a few specialized applications.

There are two kinds of programs that translate 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 translates the program a little at a time, alternately reading lines and carrying out commands.



A compiler reads the program and translates it all at once, before executing any of the commands. Often you compile the program as a separate step, and then execute the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.



Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: command-line mode and script mode. In command-line mode, you type Python statements and the interpreter prints the result.

```
$ python
Python 1.5.2 (#1, Feb 1 2000, 16:32:16)
Copyright 1991-1995 Stichting Mathematish Centrum, Amsterdam
>>> print 1 + 1
2
```

The first line of this example is the command that starts the Python interpreter. The next two lines are messages from the interpreter. The third line starts with `>>>`, which is the prompt the interpreter uses to indicate that it is ready. We typed `1+1` and the interpreter replied `2`.

Alternatively, we can write a program in a file and use the interpreter 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 `latoya.py` with the following contents:

```
print 1 + 1
```

By convention, files that contain Python programs have names that end with `.py`.

To execute the program, you have to tell the interpreter the name of the script.

```
$ python latoya.py
2
```

In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.

Most of the examples in this book are executed on the command line. Working on the command line is convenient for program development and testing, since you can type Python statements and execute them immediately. Once you have a working program, you will want to store it in a script so you can execute or modify it in the future.

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, like solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program.

The instructions (or commands, or statements) look different in different programming languages, but there are a few basic functions that appear in just about every language:

input: Get data from the keyboard, or 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, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until

eventually the subtasks are simple enough to be performed with one of these simple 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 since 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**.

There are three kinds of errors that can occur in a program. It is useful to distinguish between them in order to track them down more quickly.

1.3.1 Syntax errors

Python 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 your 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. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python 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 find them faster.

1.3.2 Run-time errors

The second type of error is a run-time 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.

For the simple sorts of programs we will be writing for the next few weeks, run-time errors are rare, so it might be a little while before you encounter one.

1.3.3 Logic errors

The third type of error is the **logical** or **semantic** error. If there is a logical 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

logical errors can be tricky, since it requires you to work backwards 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 in this class 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 lead 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." (from A. Conan Doyle's *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 always start with a working 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 Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux" (from *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, like 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 specific 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. Also, H_2O is a syntactically correct chemical name, but ${}_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and 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 I know). Similarly, ${}_2Zz$ is not legal because there is no element with the abbreviation 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 have a plus sign immediately after an equals 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 unconsciously). 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, that is, 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 (everyone) 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 effect or 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, remember that 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 people write in a new language is called "Hello, World!" because all it does is print the words "Hello, World!" In Python, this program looks like this:

```
print "Hello, world!"
```

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

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 Python 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 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 translating the program.

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

byte code: A special kind of object code used for Python programs. Byte code is similar to a low-level language, but it is portable, like a high-level language.

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 errors.

syntax: The structure of a program.

semantics: The meaning of a program.

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

run-time error: An error that does not occur until the program has started to execute, but that prevents the program from continuing.

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

formal language: Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

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

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

Chapter 2

Variables, expressions, and statements

2.1 Values

A value is one of the fundamental things – like a letter or a number – that a program manipulates. The only values we have seen so far are the result of adding `1 + 1` and the string value, `"Hello, World!"`. You (and the interpreter) can identify string values because they are enclosed in quotation marks.

There are other kinds of values, including integers and decimal numbers. An integer is a whole number like 1 or 17. Decimal numbers are numbers that have a decimal point, like 1.0 or 3.14159. You can output integer and decimal values the same way you output strings:

```
>>> print 4
4
>>> print 2.17
2.17
```

Every value has a type. If you are not sure what type a value has, you can ask the Python interpreter.

```
>>> type("Hello, World!")
<type 'string'>
>>> type(17)
<type 'int'>
>>> type(3.2)
<type 'float'>
```

Not surprisingly, the integer type is called `int`. The decimal type is called `float` because these numbers are represented in a format called **floating-point**.

The values "17" and "3.2" are strings, because they are enclosed in quotation marks. It doesn't matter that the contents of the string happen to be digits.

```
>>> type("17")
<type 'string'>
>>> type("3.2")
<type 'string'>
```

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is a legal expression.

```
>>> print 1,000,000
1 0 0
```

Well, that's not what we expected at all! It turns out that 1,000,000 is a tuple, something we'll get to in a few chapters. For now, just remember not to put commas in your integers.

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.

To create a new variable, you name it and specify the value you want it to refer to. The statement that does that is called an **assignment** because it assigns a value to a variable.

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

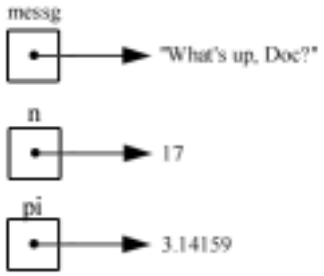
This example shows three assignments. The first assigns the value "What's up, Doc?" to a new variable named messg. The second gives the value 17 to n, and the third gives the value 3.14159 to pi.

Just as values have types, so do variables.

```
>>> type(messg)
<type 'string'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

In each case the type of the variable is the type of the value that is assigned to it.

A common way to represent variables on paper is to draw a box with the name of the variable on the outside. The box contains an arrow that points to the value of the variable. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (you can think of it as the variable's "state of mind"). This diagram shows the effect of the three assignment statements above:



2.3 Variable names and keywords

By convention, most variable names in Python contain only lower case letters. Also, most of the time programmers choose names for their variables that are meaningful – they document what the variable is used for.

But their are some rules about what is or is not a legal name in Python.

1. Names can be arbitrarily long.
2. Names can contain letters and numbers, but the first character has to be a letter.
3. Names can contain upper and lower case letters.

Upper and lower case letters are different, so `bruce` and `Bruce` are different variable names. The underscore character, _, is also legal, and is often used to separate names with multiple words, like `my_name` or `price_of_tea_in_china`.

If you try to give a variable an illegal name, you will get a syntax error.

```
>>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

`76trombones` is illegal because it does not begin with a letter. `more$` is illegal because it contains an illegal character, the dollar sign. But what's wrong with `class`?

It turns out that class is one of the Python **keywords**. Keywords are used by the language to define its rules and structure, and they can not be used as variable names.

Python has 28 keywords:

```
and       continue  else      for       import    not       raise
assert    def       except    from      in        or        return
break     del       exec      global    is        pass      try
class     elif      finally   if        lambda    print     while
```

You might want to keep this list handy. If the interpreter complains about one of your variable names, and you don't know why, see if it is on this list.

2.4 Printing variables

We have already used the print statement to display values; we can also use it to display the value assigned to a variable.

```
>>> print messg
What's up, Doc?
>>> print n
7
>>> print pi
3.14159
```

The print statement works both on the command line and in scripts. On the command line there is a more concise option; you can just type the name of the variable.

```
>>> messg
"What's up, Doc?"
```

In a script, this is a legal statement, but it does not do anything (try it).

The print statement can print multiple values on a single line:

```
>>> print "The value of pi is", pi
The value of pi is 3.14159
```

The comma separates the list of values and variables that are printed. Notice that there is a space between the values.

2.5 Operators and expressions

Operators are special symbols that represent simple computations like addition and multiplication. The values that the operator uses are called **operands**. Many of the of the operators in Python do exactly what you would expect

them to do, because they are common mathematical symbols. For example, the operator for adding two integers is +.

The following are all legal Python expressions whose meaning is more or less clear:

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

The symbols +, -, /, and the use of parenthesis for grouping are each used the same way that they are in mathematics. The * is the symbol for multiplication, and ** is the symbol for exponentiation.

Expressions can contain variable names as well as values. In each case the name of the variable is replaced with its value before the computation is performed.

Addition, subtraction, multiplication, and exponentiation all do what you expect, but you might be surprised by division. For example, the following program:

```
hour = 11
minute = 59
print "Number of minutes since midnight: ", hour*60+minute
print "Fraction of the hour that has passed: ", minute/60
```

would generate the following output:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

The first line is what we expected, but the second line is odd. The value of the variable `minute` is 59, and 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing `integer division`.

When both of the operands are integers, the result must also be an integer, and by convention integer division always rounds *down*, even in cases like this where the next integer is very close.

A possible alternative in this case is to calculate a percentage rather than a fraction:

```
print "Percentage of the hour that has passed: ", minute*100/60
```

The result is:

```
Percentage of the hour that has passed: 98
```

Again the result is rounded down, but at least now the answer is approximately correct. Another alternative is to use floating-point division, which we will get to in the next chapter.

2.6 Order of operations

When more than one operator appears in an expression the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym **PEMDAS** is a useful way to remember the order of operations:

- **P**arenthesis have the highest precedence and can be used to force an expression to evaluate in the order that you want it to. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.
- **E**xponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.
- **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So `2*3-1` yields 5 rather then 4, and `2/3-1` is `-1`, not `1` (remember that in integer division `2/3` is `0`).
- Operators with the same precedence are evaluated from left to right. So in the expression `minute*100/60`, the multiplication happens first, yielding `5900/60`, which in turn yields `98`. If the operations had gone from right to left, the result would be `59/1` which is `59`, which is wrong.

2.7 Operations on strings

In general you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that `messg` has type `string`)

```
messg-1   "Hello"/123   messg*"Hello"   "15"+2
```

Interestingly, the `+` operator *does* work with strings, although it does not do exactly what you might expect. For strings, the `+` operator represents **concatenation**, which means joining up the two operands by linking them end-to-end. For example,

```
fruit = "banana"
bakedGood = " nut bread"
dessert = fruit + bakedGood
print dessert
```

The output of this program is `banana nut bread`.

The `*` operator also works on strings; it performs **repetition**. For example, `'Fun'*3` is `'FunFunFun'`. One of the operands has to be a string; the other has to be an integer.

On one hand, this interpretation of + and * makes sense by analogy with addition and multiplication. Just as 4*3 is equivalent to 4+4+4, we expect "Fun"*3 to be the same as "Fun"+"Fun"+"Fun", and it is. On the other hand, there is a significant way in which string concatenation and repetition are very different from integer addition and multiplication.

As an exercise, name a property that addition and multiplcation have that string concatenation and repetition do not.

2.8 Composition

So far we have looked at the elements of a program – variables, expressions, and statements–in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to add numbers and we know how to print; it turns out we can do both at the same time:

```
>>> print 17 + 3
20
```

Actually, we shouldn't say "at the same time," since in reality the addition has to happen before the printing, but the point is that any expression, involving numbers, strings, and variables, can be used inside a print statement. We've already seen an example of this:

```
print "Number of minutes since midnight: ", hour*60+minute
```

And you can also put arbitrary expressions on the right-hand side of an assignment statement:

```
percentage = (minute * 100) / 60
```

This ability may not seem impressive now, but we will see other examples where composition makes it possible to express complex computations neatly and concisely.

WARNING: There are limits on where you can use certain expressions. For example, the left-hand side of an assignment statement has to be a *variable* name, not an expression. So the following is illegal: `minute+1 = hour`.

2.9 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense and it often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason it is a good idea to add notes to your programs to explain, in natural language, what the program is doing. These notes are called **comments** and they are marked with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60     # caution: integer division
```

Everything from the # to the end of the line is ignored—it has no effect on the program. The message is intended for the programmer, or for future programmers that might have to use this code. In this case it reminds the reader about the ever-surprising behavior of integer division.

2.10 Glossary

value: A number or string (or other thing to be named later) that can be stored in a variable or computed in an expression.

variable: A name that refers to a value.

type: A set of values. The type of a value determines how it can be used in expressions. So far, the types we have seen are integers (type `int`), floating point numbers (type `float`) and strings (type `string`).

keyword: A reserved word that is used by the compiler to parse programs. You cannot use keywords, like `if`, `def` and `while` as variable names.

statement: A line of code that represents a command or action. So far, the statements we have seen are assignments and print statements.

assignment: A statement that assigns a value to a variable.

comment: A piece of information in a program which is meant for other programmers (or anyone reading the source code) and which has no effect on the execution of the program.

state diagram A graphical representation of a set of variables and the values they refer to.

expression: A combination of variables, operators and values that represents a single result value.

operator: A special symbol that represents a simple computation like addition, multiplication or string concatenation.

operand: One of the values on which an operator operates.

integer division: An operation which divides one integer by another and returns an integer. Integer division returns only the whole number of times that the numerator is divisible by the denominator and discards any remainder.

rules of precedence: The set of rules governing the order in which expressions involving multiple operators and operands will be evaluated.

concatenate: Join two operands end-to-end.

composition: The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

Chapter 3

Functions

3.1 Function calls

We have already seen one example of a **function call**:

```
>>> type("32")
<type 'string'>
```

The name of the function is `type`, and it displays the type of a value or variable. The value or variable, which is called the **argument** of the function, has to be enclosed in parentheses. 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.

```
>>> betty = type("32")
>>> print betty
<type 'string'>
```

As another example, the `id` function takes a value or a variable and returns an integer that acts as a unique identifier for the value.

```
>>> id(3)
134882108
>>> betty = 3
>>> id(betty)
134882108
```

Every value has an `id`. The `id` of a variable is the `id` of the value it refers to.

3.2 Type conversion

Python provides a collection of built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if possible, or complains otherwise:

```
>>> int("32")
32
>>> int("Hello")
ValueError: invalid literal for int(): Hello
```

`int` can also convert floating-point values to integer, but remember that it always rounds down:

```
>>> int(3.99999)
3
```

There is also a `float` function that converts integers and strings:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
```

Finally, there is the `str` function, which converts to type `string`:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

It may seem odd that Python distinguishes the integer value `1` from the floating-point value `1.0`. They may represent the same number, but they belong to different types. The reason is that they are represented differently inside the computer. The details of this representation are not important for now, but they affect the behavior of some programs.

3.3 Type coercion

Now that we can convert between types, we can solve the problem we had in the last chapter with integer division. We were trying to calculate the fraction of an hour that had elapsed. The most obvious expression, `minute / 60`, does integer arithmetic, which is not what we want.

One alternative is to convert `minute` to floating-point and do floating-point division:

```
>>> minute = 59
>>> float(minute) / 60.0
0.983333333333
```

Or we can take advantage of Python's type **coercion**. For the mathematical operators, if either operand is a `float`, the other is automatically converted to a `float`.

```
>>> minute = 59
>>> minute / 60.0
0.983333333333
```

By making the denominator a `float`, we force Python 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 we evaluate the argument of the innermost function, then evaluate the function, and so on.

Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions grouped together.

Before we can use the functions from a module, we have to import them:

```
import math
```

To call one of the functions, we have to specify the name of the module and the name of the function, separated by a dot:

```
decibel = math.log10 (17.0)
angle = 1.5
height = math.sin(angle)
```

The first statement sets ¡tt¿decibel¡/tt¿ to the logarithm of 17, base `10`. There is also a function called `log` that takes logarithms base `e`.

The third statement finds the sine 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, you can 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 sine:

```
degrees = 45
angle = degrees * 2 * math.pi / 360.0
math.sin(angle)
```

The constant `pi` is also part of the math module. If you know your geometry, you can verify the result by comparing it to the square root of 2 divided by 2:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.5 Composition

Just as with mathematical functions, Python 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 = math.cos(angle + pi/2)
```

This statement takes the value of ¡tt¿pi¡/tt¿, divides it by two 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 = math.exp(math.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 are built into Python, but it is also possible to add new functions.

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 name you want for your function, except that you can't use a name that is a Python 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. In the examples in this book, we will use an indentation of two spaces.

The first couple of functions we are going to write have no parameters, so the syntax looks like this:

```
def new_line():
  print
```

This function is named `new_line`. 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).

We can call the new function using the same syntax we use for built-in functions:

```
print "First Line."
new_line()
print "Second Line."
```

The output of this program is

```
First line.

Second line.
```

Notice the extra space between the two lines. What if we wanted more space between the lines? We could call the same function repeatedly:

```
print "First Line."
new_line()
new_line()
new_line()
print "Second Line."
```

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

```
def threeLines():
  new_line()
  new_line()
  new_line()

print "First Line."
threeLines()
print "Second Line."
```

This function contains three statements, all of which are indented by two spaces. Since the next statement is not indented, Python 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 `new_line`.

So far, it may not be clear why it is worth the trouble to create all these new functions. Actually, there are a lot of reasons, but this example demonstrates two:

- Creating a new function gives you an opportunity to give a name to 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.
- 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. How would you print 27 new lines?*

3.7 Definitions and use

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def new_line():
  print

def threeLines():
  new_line()
  new_line()
  new_line()

print "First Line."
threeLines()
print "Second Line."
```

This program contains two function definitions: `new_line` and `threeLines`. Function definitions get executed just like other statements, but the effect is to create the new function. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. It other words, the function definition has to be executed before the first time it is called.

As an exercise, try running this program with the last three statements moved to the top of the program and record which error message you get.

As another exercise, try taking the working version of the program and moving the definition of `new_line` *after the definition of* `threeLines`*. What happens when you run this program?*

3.8 Flow of execution

In order to insure 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. Statements are executed one at a time, in order, unless you reach a function call.

Function definitions do not alter the flow of execution of the program, but remember that 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 we are in the middle of one function, we might have to go off and execute the statements in another function. But while we are executing that new function, we might go off and execute yet another function!

Fortunately, Python 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 we have used require arguments, the values that control how the function does its job. For example, if you want to find the sine 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, like `pow`, which 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 a parameter:

```
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) gets 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')
Spam Spam
>>> printTwice(5)
5 5
>>> printTwice(3.14159)
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 `float`.

The same rules of composition that apply to built-in functions also apply to user-defined functions, so you can use any kind of expression as an argument for `printTwice`.

```
>>> printTwice('Spam'*4)
SpamSpamSpamSpam SpamSpamSpamSpam
>>> printTwice(math.cos(math.pi))
-1.0 -1.0
```

As usual, the expression is evaluated before the function is run.

Also, we can use a variable as an argument:

```
>>> latoya = 'Eric, the half a bee.'
>>> printTwice(latoya)
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 (`latoya`) 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`.

3.10 Variables and parameters are local

When you create a variable inside a function, it only exists inside the function, and you cannot use it outside. For example, the function

```
>>> def catTwice(part1, part2):
      cat = part1 + part2
>>> printTwice(cat)
```

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

```
>>> chant1 = "Die Jesu domine, "
>>> chant2 = "Dona eis requiem."
>>> catTwice(chant1, chant2)
Die Jesu domine, Dona eis requiem. Die Jesu domine, Dona eis requiem.
```

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

```
>>> print cat
NameError: cat
```

Similarly, if we try to use `cat` inside `printTwice` we get an error:

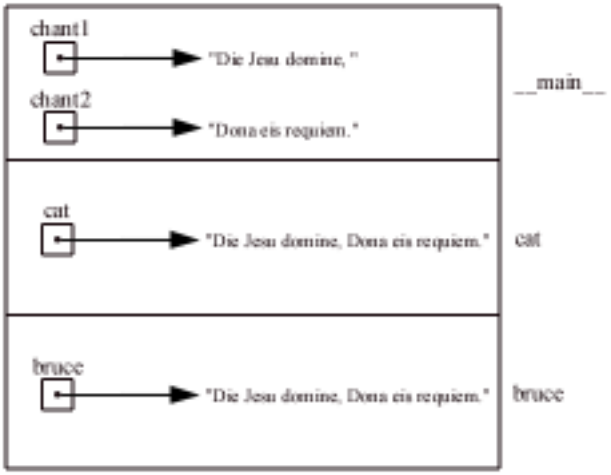
```
>>> def printTwice(bruce):
...   print cat, cat

>>> catTwice(chant1, chant2)
NameError: cat
```

The same rules that apply to variables also apply to parameters. For example, outside the function `printTwice`, there is no such thing as `bruce`. If you try to use it, Python will complain.

3.11 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 each variable belongs to.

Each function is represented by a box with the name of the function beside it. The parameters and variables that belong to that function go inside. For example, the stack diagram for the previous program 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__`.

In each case, a parameter refers to the same value as the corresponding argument. So `part1` in `catTwice` has the same value as `chant1` in `__main__`.

3.12 Functions with results

You might have noticed by now that some of the functions we are using, like the math functions, yield results. Other functions, like `new_line`, 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, like `new_line() + 7`?

3. Can we write functions that yield results, or are we stuck with things like `new_line` and `printTwice`?

The answer to the third question is "yes, you can write functions that return values," and we'll do it in Chapter 5.

> *As a final exercise, answer the other two questions by trying them out. Any time you have a question about what is legal or illegal in Python, a good way to find out is to ask the interpreter.*

3.13 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 which is sent back to the part of the program from which the function was called. If a function call appears on the left hand side of an assignment statement and a variable on the right hand side, then the return value will be referenced by the variable.

coercion: The forced conversion of values of one mathematical type into equivalent values of another mathematical type to make evaluation of expressions involving mixed types possible. For example, in the expression `3/2.0`, `3` is of type `int`, and `2.0` is of type `float`, so `3` must first be *coersed* into a `float` before floating point division can be performed.

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

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 variables can only be used inside its function.

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

Chapter 4

Conditionals and recursion

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 Python, the modulus operator is a percent sign, `%`. The syntax is the same as for other operators:

```
quotient = 7 / 3
remainder = 7 % 3
```

The first operator, integer division, yields 2. The second operator yields 1. Thus, 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 use the modulus operator to extract the rightmost digit or digits from a number. For example, `x % 10` yields the rightmost digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

4.2 Conditional execution

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

```
if x > 0:
  print "x is positive"
```

The expression between the `if` and the `:` is called the **condition**. If it is true, then the indented statement gets executed. If the condition is not true, nothing happens.

The condition can contain any of the *comparison operators*:

```
x == y              # x equals y
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 syntax Python uses is a little different from symbols used in mathematics. A common error is to use a single = instead of a double ==. Remember that = *is the assignment operator*, and == *is a comparison operator.* Also, there is no such thing as =< or =>.

4.3 Compound Statements

The `if` statement is the second example we have seen of a **compound statement**. The first was the function definition. All compound statements have a common syntax:

```
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 statement **block** or statement **body**. The first unindented statement marks the end of the block.

All the statements in the body are treated as a unit. In the case of an `if` statement, either all of them or none of them are executed.

4.4 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:

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

If the remainder when `x` is divided by 2 is zero, then we know that `x` is even, and this 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.

As an aside, if you think you might want to check the parity (evenness or oddness) of numbers often, you might want to "wrap" this code up in a function:

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

Now you have a function named `printParity` that displays an appropriate message for any integer you care to provide. You would call this function as follows:

```
printParity(17)
```

4.5 Multiple Branches

Another name for conditional execution is **conditional branching**, because this type of control structure causes the flow of execution to branch off in different directions. Sometimes you want to check for a number of related conditions and choose one of several actions. The following trichotomy test shows how to do this multiple branching:

```
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"
```

As an exercise, wrap the trichotomy code above in a function called `compare(x, y)`.

The `elif` statement ("elif" is an abbreviation of "else if") can be repeated as many times as needed to select from several conditions:

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

4.6 Nested conditionals

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

```
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
```

There is now an outer conditional that contains two branches. The first branch contains a simple output statement, but the second branch contains another `if` statement, which has two branches of its own. Fortunately, 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 get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

On the other hand, this kind of **nesting** is common, and we will see it again, so you should become familiar with it.

4.7 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 math

def printLogarithm(x):
  if x <= 0:
    print "Positive numbers only, please."
    return

  result = math.log(x)
  print "The log of x is", result
```

This defines a function named `printLogarithm` that takes a parameter named `x`. The first thing it does is check whether `x` is less than or equal to zero, 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 any time you want to use a function from the math module, you have to import it.

4.8 Recursion

We mentioned in the last chapter that it is legal for one function to call another, and we 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):
  if n == 0:
    print "Blastoff!"
  else:
    print n
    countdown(n-1)
```

The name of the function is `countdown` and it takes a single parameter. If the parameter is zero, it outputs the word "Blastoff." Otherwise, it outputs the parameter 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 zero, it outputs the value 3, and then calls itself...

> The execution of `countdown` begins with `n=2`, and since `n` is not zero, it outputs the value 2, and then calls itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is not zero, it outputs the value 1, and then calls itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is zero, 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:

```
3
2
1
Blastoff!
```

As a second example, let's look again at the functions `newLine` and `threeLine`.

```
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

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

This program is similar to `countdown`; as long as `n` is greater than zero, 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 called **recursion**, and such functions are said to be **recursive**.

4.9 Infinite recursion

In the examples in the previous section, each time the functions get called recursively, the argument gets smaller by one, so eventually it gets to zero. When the argument is zero, the function returns immediately, *without making any recursive calls*. This case—when the function completes without making a recursive call—is called the **base case**.

If a recursion never reaches a base case, it will go on making recursive calls forever and the program will never terminate. This is known as **infinite recursion**, and it is generally not considered a good idea.

In most programming environments, a program with infinite recursion will not really run forever. Instead, the maximum recursion depth will be reached and Python will report an error message:

```
RuntimeError: Maximum recursion depth exceeded
```

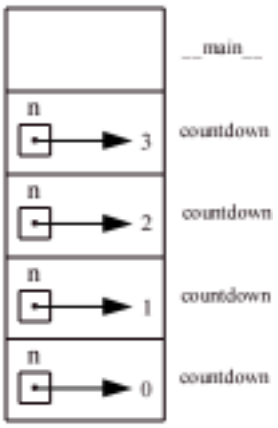
> *As an exercise, write a function with infinite recursion and run it in the Python interpreter so that you get the error message above.*

4.10 Stack diagrams for recursive functions

In the previous chapter we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can make it easier to interpret a recursive function.

Every time a function gets called, it creates a new instance of the function. The instance contains the function's local variables and parameters. When the function completes, that instance goes away. For a recursive function, there might be more than one instance on the stack at the same time.

This figure shows a stack diagram for countdown, called with `n = 3`:



There is one instance of `__main__` and four instances of `countdown`, each with a different value for the parameter `n`. The bottom of the stack, `countdown` with `n=0`, is the base case. It does not make a recursive call, so there are no more instances of `countdown`.

The instance of `__main__` is empty because we did not create any variables in it or pass any parameters to it.

> *As an exercise, draw a stack diagram for* `nLines`*, invoked with the parameter* `n=4`*.*

4.11 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.

Python provides built-in functions that get input from the keyboard. The simplest is called `raw_input`. When you call this function, the program stops and waits for the user to type something. When the user presses `return` or `enter`, the program resumes and `raw_input` return what the user types as a `string`:

```
>>> input = raw_input ()
What are you waiting for?
>>> print input
What are you waiting for?
```

Before calling `raw_input`, it is a good idea to print a message telling the user

what to input. This message is called a **prompt**. You can supply a prompt as an argument to `raw_input`:

```
>>> name = raw_input ("What...is your name? ")
What...is your name? Arthur, King of the Britons!
>>> print name
Arthur, King of the Britons!
```

If you are expecting the response to be an integer, you can use the `input` function. For example,

```
>>> prompt = "What...is the airspeed velocity of an unladen swallow?\n"
>>> speed = input (prompt)
```

If the user types a string of digits, it will be converted to an integer and assigned to `speed`. Unfortunately, if the user types a non-digit, the program crashes:

```
>>> speed = input (prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
SyntaxError: invalid syntax
```

To avoid this kind of error, it is generally a good idea to use `raw_input` to get a string and then use the conversion functions to convert to other types.

4.12 Glossary

modulus operator: An operator that works on integers and yields the remainder when one number is divided by another. In Python it is denoted with a percent sign (%).

conditional statement: A statement that controls the flow of execution in a program depending on some condition.

compound statement: A Python that consists of a header ending with a colon and a block or body of one or more statements with the same indentation.

block: A group of one or more statements that are treated as a single statement in the sense that they are all executed together in sequence. In Python the block of statements which make up the body of a compound statement must all be indendent the same amount relative to the statement header.

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

conditional branching: A flow of execution that can follow several possible paths, depending on the effect of condition statements.

nesting: One program structure within another; for example, a conditional statement inside one or both branches of another conditional statement.

recursion: The process of calling the function you are currently executing.

base case: The branch of the conditional statement in a recursive function that does not result in a recursive call. If a call to a recursive function does not eventually lead to the base case than it will result in infinite recursion.

infinite recursion: A function that calls itself recursively without every reaching the base case. Eventually an infinite recursion will cause 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, like the math functions, have produced results. That is, the effect of calling the function is to generate a new value, which we usually assign to a variable or use as part of an expression. For example:

```
import math

e = math.exp(1.0)
height = radius * math.sin(angle)
```

But so far none of the functions we have written have returned a value. When you call a function that does not return a value, it is typically on a line by itself, with no assignment:

```
nLines(3)
countdown(n-1)
```

In this chapter, we are going to write functions that return things, which we will refer to as 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 math

def area(radius):
  temp = math.pi * radius**2
  return temp
```

In the last line of the function the `return` statement now includes a return value. This statement means, "return immediately from this function and use the following expression as a return value." The expression you provide can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):
  return math.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):
  if x < 0:
    return -x
  else:
    return x
```

Since these `return` statements are in an alternative conditional, only one will be executed. Although it is legal to have more than one `return` statement in a function, you should keep in mind that as soon as one is executed, the function terminates without executing any subsequent statements.

Code that appears after a `return` statement, or any place else where it can never be executed, is called **dead code**. It can never be executed because it can never be reached by any possible flow of execution of the program.

If you put return statements inside a conditional, then you have to guarantee that *every possible path* through the program hits a return statement. For example:

```
def absoluteValue(x):
  if x < 0:
    return -x
  elif x > 0:
    return x           # ERROR!!
```

This program is not correct because if `x` happens to be 0, then neither condition will be true and the function will end without hitting a return statement. In this case, the return value is a special value called **None**.

```
>>> print absoluteValue(0)
None
```

As an exercise, rewrite the `compare` *function that you wrote in section 4.5 so that it 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 Python functions and tell what they do. You also have some experience making small modifications to

existing functions. It may not be clear to you yet, however, how to go about writing a function from scratch.

We are going to suggest one technique that we will call **incremental development**.

As an example, imagine you want to find the distance between two points, given by the coordinates (X_1, y_1) and (x_2, y_2). By the usual definition, with `sqrt` representing the square root function,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

The first step is to consider what a `distance` function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value).

In this case, the two points are the parameters, which we can represent using four parameters. The return value is the distance.

Already we can write an outline of the function:

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

The `return` statement is a placekeeper so that the function will run when we test it and return something, even though it is not the right answer. At this stage the function doesn't do anything useful, but it is worthwhile to try running it so we can identify any syntax errors before we make it more complicated.

In order to test the new function, we have to call it with sample values, as in the following session:

```
>>> def distance(x1, y1, x2, y2):
...   return 0.0
...
>>> distance(1, 2, 4, 6)
0.0
>>>
```

These values were chosen so that the horizontal distance is 3 and the vertical distance is 4; that way, the result will be 5 (the hypotenuse of a 3-4-5 triangle). When you are testing a function, it is useful to know the right answer.

Once we have checked the syntax of the function definition, we can start adding lines of code one at a time. After each incremental change, we test the function again. That way, at any point we know exactly where the error must be—in the last line we added.

The next 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`.

```
def distance(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
```

```
    print "dx is", dx
    print "dy is", dy
    return 0.0
```

We added output statements that will let us to check the intermediate values before proceeding. We already know that they should be 3 and 4.

When the function is finished we will remove the output statements. Code like that is called **scaffolding**, because it is helpful for building the program, but it is not part of the final product. Sometimes it is a good idea to keep the scaffolding around, but comment it out, just in case you need it later.

The next step in the development is to square `dx` and `dy`.

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

Again, we would run the program at this stage and check the intermediate value (which should be 25).

Finally, if we have imported it from the math module, we can use the `sqrt` function to compute and return the result.

```
def distance(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  dsquared = dx**2 + dy**2
  result = sqrt(dsquared)
  return result
```

As you gain more experience programming, you might find yourself writing and debugging more than one line at a time. Nevertheless, this 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. You should record each stage of the incremental development process as you go.*

5.3 Composition

As you should expect by now, once you define a new function, you can use it as part of an expression, and you can build new functions using existing functions. For example, what if someone gave you two points, the center of the circle and a point on the perimeter, and asked for the area of the circle?

Let's say 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 have 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 all up in a function, we get:

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

We called this function ¡tt¿area2¡/tt¿ to distinguish it from the `area` function defined earlier. There can only be one function of a given name within a given module. We will talk more about this later when we discuss modules and namespaces.

The temporary variables `radius` and `area` 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 expressions and logical operators

The condition statement that follows an `if` is an example of a **boolean expression**. Boolean expressions are expressions which evaluate to either true or false. In Python, boolean expressions evaluate to `1` if the expression is true and `0` if it is false. For example:

```
>>> 5 == 5
1
>>> 5 == 6
0
>>>
```

The operator == compares two values and produces a boolean value. In the first statement the two operands are equal, so the expression evaluates to `1`; in the second example, 5 is not equal to 6, so we get `0`.

There are three **logical operators** in Python: `and`, `or` and `not`. 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 zero 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 has the effect of negating or inverting 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.

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 of the conditionals, so we need to use the `and` operator:

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

These kinds of conditions are common, so Python provides an alternate syntax that is similar to mathematical notation:

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

Python evaluates the expressions involving the operands on both sides of each operator and `and`s them together to produce a result.

5.5 Boolean functions

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

```
def isDivisible(x, y):
  if x % y == 0:
    return 1       # it's true
  else:
    return 0       # it's false
```

The name of this function is `isDivisible`. It is common to give boolean functions names that sound like yes/no questions. We return either `1` or `0` to indicate whether the argument passed to it is or isn't a single digit.

We can reduce the size of this function by taking advantage of the fact that the conditional statement after the `if` is itself a boolean expression. We can simply return it directly, avoiding the `if` statement altogether:

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

The following session shows the new function in action:

```
>>> isDivisible(6, 4)
0
>>> isDivisible(6, 3)
1
```

The most common use of boolean functions is inside conditional statements

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

As an exercise, write a function `isBetween(x, y, z)` *that returns* `1` *whenever* `y <= x <= z` *and* `0` *otherwise.*

5.6 More recursion

So far we have only learned a small subset of Python, but you might be interested to know that this subset is now a *complete* programming language, by which we mean that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features we have used so far (actually, we would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that claim is a non-trivial exercise first accomplished by Alan Turing, one of the first computer scientists (well, some would argue that he was a mathematician, but a lot of the 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 we 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 in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function, `factorial`, you might get something like:

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

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, we get 3! equal to 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a Python 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 zero, all we have to do is return 1:

```
def factorial(n):
  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.

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

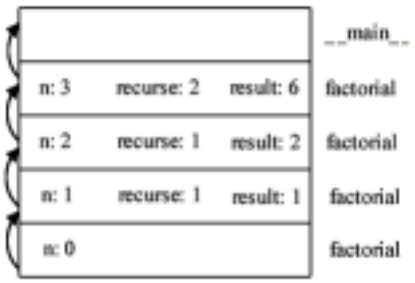
If we look at the flow of execution for this program, it is similar to `nLines` from the previous chapter. If we call `factorial` with the value 3:

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

> Since 2 is not zero, we take the second branch and calculate the factorial of $n - 1$...
>
> > Since 1 is not zero, we take the second branch and calculate the factorial of $n - 1$...
> >
> > > Since 0 *is* zero, we take the first branch and return the value 1 immediately without making any more recursive calls.
> >
> > The return value (1) gets multiplied by n, which is 1, and the result is returned.
>
> The return value (1) gets multiplied by n, which is 2, and the result is returned.

The return value (2) gets multiplied by n, which is 3, and the result, 6, is returned to `__main__`, or whoever called `factorial (3)`.

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.

Notice that in the last instance of `factorial`, the local variables `recurse` and `result` do not exist because when $n = 0$ the branch that creates them does not execute.

5.7 Leap of faith

Following the flow of execution is one way to read programs, but as you saw in the previous section, 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 `math.cos` or `math.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.

Well, the same is true when you call one of your own functions. For example, in Section 5.5 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 examination of the code—we can use the function without ever 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 is a bit strange to assume that the function works correctly when you have not even finished writing it, but that's why it's called a leap of faith!

5.8 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):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

From now on we will tend to use the more concise version, 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 `fibonacci`, which has the following definition:

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

Translated into Python, this is

```
def fibonacci (n):
  if n == 0 or 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 we assume that the two recursive calls (yes, you can make two recursive calls) work correctly, then it is clear that we get the right result by adding them together.

5.9 Checking types

What happens if we call factorial and give it 1.5 as an argument?

```
>>> factorial (1.5)
RuntimeError: Maximum recursion depth exceeded
```

It looks like an infinite recursion. But how can that be? We have a base case, when $n == 0$. The problem is that we are missing the base case. In the first recursive call, the value of n is 0.5. In the next instance, it is -0.5. From there it gets smaller and smaller, but it will never be 0.

We have two choices. We can try to generalize the factorial function so that it works with floating-point numbers, or we can make factorial check the type of its parameter. The first option has been done; it's called the Gamma function. So we'll go for the second.

We can use the `type` function to compare the type of the parameter to the type of a known integer value (like 1). While we're at it, we can make sure the parameter is positive:

```
def factorial (n):
  if type(n) != type(1):
    print "Factorial is only defined for integers."
    return -1
  elif n < 0:
    print "Factorial is only defined for positive integers."
    return -1
  elif n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

Now, in effect, we have three base cases. The first catches non-integers. The second catches negative integers. In both cases we print an error message and then return a special value -1 to indicate to the caller that something went wrong.

```
>>> factorial (1.5)
Factorial is only defined for integers.
-1
>>> factorial (-2)
Factorial is only defined for positive integers.
```

```
-1
>>> factorial ("fred")
Factorial is only defined for integers.
-1
```

If we get past both checks, then we know that `n` is a positive integer, and so we can prove that the recursion terminates.

This style of programming is sometimes called a **guardian** pattern. The first two conditionals act as guardians, protecting the code the follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

5.10 Glossary

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

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

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

None: A special Python value which is returned by functions that either do not have a return statement or have a return statement without an argument. It is also possible to assign this value to a variable as in the statement: `x = None`. It is equivalent to a false boolean or an empty list.

incremental development: A program development methodology that uses small, incremental changes to a working program to gradually modify it until it does what it is intended to do.

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

boolean expression: An expression that evaluates to one of two states, often called `true` and `false`.

comparison operator: An operator that compares two values and produces a boolean value that indicates the relationship between the operands. The comparison operators in Python are `==`, `!=`, `>`, `<`, `>=`, and `<=`.

logical operator: An operator that combines boolean values in order to test compound conditions. The logical operators in Python are `and`, `or`, and `not`.

guardian: A condition that checks for and handles circumstances that might cause an error.

Chapter 6

Iteration

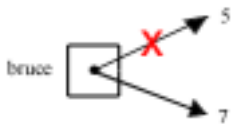
6.1 Multiple assignment

We haven't discussed it until now, but it is legal in Python to make more than one assignment to the same variable. The effect of the new assignment is to redirect the variable so that it stops referring to the old value and starts referring to the new value.

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

The output of this program is `5 7`, because the first time we print `bruce` his value is 5, and the second time his value is 7. The comma at the end of the first `print` statement stops a newline from being printed at that point.

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



When there are multiple assignments to a variable, it is especially important to distinguish between an assignment statement and a statement of equality. Because Python uses the `=` symbol for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First of all, equality is commutative, and assignment is not. For example, in mathematics if $a = 7$ then $7 = a$. But in Python the statement `a = 7` is legal, and `7 = a` is not.

Furthermore, in mathematics, a statement of equality is true for all time. If $a = b$ now, then a will always equal b. In Python, 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 it does not change the value of `b`, and so they are no longer equal. In some programming languages an alternate symbol is used for assignment, such as `<-` or `:=`, in order to avoid confusion.

Although multiple assignment is frequently useful, you should use it with caution. If the values of variables are changing constantly in different parts of the program, it can make the code difficult to read and debug.

6.2 Iteration

One of the things computers are often used for is the automation of repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

We have seen programs that use recursion to perform repetition, such as `nLines` and `countdown`. This type of repetition is called **iteration**, and Python provides several language features that make it easier to write iterative programs.

The first feature that we are going to look at is the `while` statement.

6.3 The `while` statement

Using a `while` statement, we can rewrite `countdown`:

```
def countdown(n):
  while n > 0:
    print n
    n = n-1
  print "Blastoff!"
```

You can almost read a `while` statement as if it were English. What this means is, "While `n` is greater than zero, continue displaying the value of `n` and then reducing the value of `n` by 1. When you get to zero, output the word 'Blastoff!"'

More formally, the flow of execution for a `while` statement is as follows:

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** of the while loop (all the statements indented the same amount under the line containing the `while`), and then go back to step 1.

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 will terminate 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 (each iteration), so eventually we have to get to zero. In other cases it is not so easy to tell:

```
def sequence(n):
  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 != 1`, so the loop will continue until `n` is `1`, which will make the condition false.

At each iteration, 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 two. If it is odd, the value is replaced by `3n+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 will terminate. 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 every time through the loop, until we get to 1. The previous example ends with such a sequence, starting with 16.

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 4.8 using iteration instead of recursion.*

6.4 Tables

One of the things loops are good for is generating tabular data. For example, before computers were readily available, people had to calculate logarithms, sines and cosines, and other common mathematical functions by hand. To

make that easier, there were books containing long tables where you could find the values of various functions. Creating these tables was slow and boring, and the result 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. It turns out that 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:

```
x = 1.0
while x < 10.0:
  print x, '\t', math.log(x)
  x = x + 1.0
```

The **escape sequence** `\t` represents a **tab** character. You can also use the escape sequence `\n` to represent a newline character. These escape sequences can be included anywhere in a string, although in this example the tab escape sequence is the only thing in the string.

A tab character causes the cursor to shift 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 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:

$$\log_2 x = \frac{log_e x}{log_e 2} \tag{6.1}$$

Changing the output statement to

```
print x, '\t',  math.log(x)/math.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', math.log(x)/math.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 we are using tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

Log 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 65536 (that's 2^{16}). Print it out and memorize it.*

6.5 Two-dimensional tables

A two-dimensional table is a table where you choose a row and a column and read the value at the intersection. A multiplication table is a good example. Let's say you wanted to print a multiplication table for the values from 1 to 6.

A good way to start is to write a simple loop that prints the multiples of 2, all on one line.

```
i = 1
while i <= 6:
  print 2*i, '\t',
  i = i + 1
print
```

The first line initializes a variable named `i`, which is going to act as a counter, or **loop variable**. As the loop executes, the value of `i` increases from 1 to 6, and then when `i` is 7, the loop terminates. Each time through the loop, we print the value `2*i` followed by three spaces. By placing a comma at the end of the `print` statement, we get all the output on a single line. After the loop completes, a `print` statement with no arguments is used to start a new line.

The output of this program is:

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

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

6.6 Encapsulation and generalization

Encapsulation usually means taking a piece of code and wrapping it up in a function, allowing you to take advantage of all the things functions are good for. We have seen two examples of encapsulation, when we wrote `printParity` in Section 4.4 and `isDivisible` in Section 5.5.

Generalization means taking something specific, like printing multiples of 2, and making it more general, like printing the multiples of any integer.

Here's a function that encapsulates the loop from the previous section and generalizes it to print multiples of `n`.

```
def printMultiples(n):
  i = 1
  while i <= 6:
    print n*i, '\t',
    i = i + 1
  print
```

To encapsulate, all we had to do was add the first 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 argument 3, the output is:

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

and with argument 4, the output is

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

By now you can probably guess how we are going to print a multiplication table: we'll call `printMultiples` repeatedly with different arguments. In fact, we are going to use another loop to iterate through the rows.

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

First of all, notice how similar this loop is to the one inside `printMultiples`. All we did was replace the print statement with a function call.

The output of this program is

```
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
```

which is a multiplication table.

6.7 More encapsulation

To demonstrate encapsulation again, we'll take the code from the end of section 6.6 and wrap it up in a function:

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

The process we are demonstrating is a common **development plan**. You develop code gradually by adding lines in __main__ or someplace else, and then when you get it working, you extract it and wrap it up in a function. This process is made even easier by making use of the interpreter. First try out your idea in the interpreter. This enables you to get immediate feedback on all those "what happens if I write this?" questions. Once you have the function working the way you want it to, use a text editor to save it in a module.

This development plan is also useful for another reason. Sometimes (more often then not when you are first learning) you don't know when you start writing exactly how to divide the program into functions. This approach lets you design as you go along.

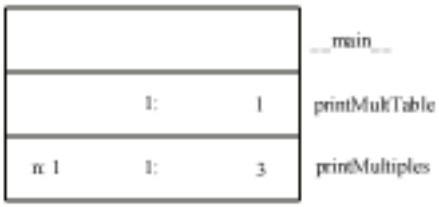
6.8 Local variables

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.*

Remember that variables created inside a function definition are local. You cannot access a local variable from outside its "home" function, and 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 clearly 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 `printMultiple` goes from 1 up to `n`. In the diagram, it happens to be 2. The next time through the loop it will be 3.

It is often a good idea to use different variable names in different functions, to avoid confusion, but there are good reasons to reuse names. For example, it is common to use the names `i`, `j` and `k` as loop variables. If you avoid using them in one function just because you used them somewhere else, you will probably make the program harder to read.

6.9 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the 6x6 table. You could add a parameter to `printMultTable`:

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

We replaced the value 6 with the parameter `high`. If `printMultTable` is called with the argument 7, we get

```
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
```

which is fine, except that we probably want the table to be square (same number of rows and columns), which means we have to add another parameter to `printMultiples`, to specify how many columns the table should have.

Just to be annoying, we will also call this parameter `high`, demonstrating that different functions can have parameters with the same name (just like local variables):

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

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

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 7x7 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 find that the resulting program has capabilities you did not intend. For example, you might notice that the multiplication table is symmetric, because $ab = ba$, so 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, follow or trace the execution of this new version of `printMultTable` *to figure out how it works.*

6.10 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 the reasons functions are useful:

- By giving a name to a sequence of statements, you make 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.

6.11 Glossary

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

iteration: The successive repetition (execution) of the body of a loop, until a terminating or exit condition is met. In its singular use an iteration refers to one pass through the loop body, including the evaluation of the condition.

body: The statements inside the loop.

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

infinite loop: A loop whose terminating condition is never satisfied.

escape sequence: An escape character (\) followed by one or more printable characters, used to designate a non-printable character. The other escape sequences are:

Escape Sequence	**Meaning**
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\n	ASCII Linefeed (LF)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
ooo	ASCII character with octal value *ooo*
\x*hh...*	ASCII character with hex value *hh...*

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

loop variable: A variable, often called a counter, that is used to determine the terminating condition of a loop. When used as a counter, a loop variable is incremented by 1 on each pass through the loop until the desired number of iterations is met.

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

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 three types: `ints`, `floats` and `strings`. Of these, `strings` are qualitatively different from the others because it is made up of smaller pieces— the characters. `strings` are an example of **compound data type**.

Depending on what we are doing, we may want to treat a compound type as a single thing, or we may want to access its parts. This ambiguity is useful.

There are a number of operations and functions we can use to access and manipulate the characters that make up a `string`. The first of these is the square brackets operator (`[` and `]`), which selects and reads a single character from a `string`:

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

The expression `fruit[1]` indicates that we want character number 1 from the string named `fruit`. The result is stored in a variable named `letter`. When we output the value of `letter`, we get a surprise:

```
a
```

`a` is not the first letter of `"banana"`. Unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter ("zero-eth") of `"banana"` is `b`. The 1th letter ("one-eth") is `a` and the 2th ("two-eth") letter is `n`.

If you want the zero-eth letter of a string, you have to put zero in the square brackets:

```
>>> letter = fruit[0]
>>> print letter
b
```

7.2 Length

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

```
>>> fruit = "banana"
>>> len(fruit)
6
```

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

```
length = len(fruit)
last = fruit[length]        # ERROR!
```

That won't work. Instead, your program will end, and you will get an error message.

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

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

Alternatively, you can use negative indices, which count backwards 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 common thing to do with a string is 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 outputs each letter on a line by itself. Notice that the condition is `index < len(fruit)`, which means that 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 we access is the one with the index `len(fruit)-1`, which is the last character in the string.

The name of the loop variable is `index`. An **index** is a variable or value used to specify one member of an ordered set, in this case the set of characters in the string. The index indicates (hence the name) which one you want.

As an exercise, write a function that takes a `string` *as an argument and that outputs the letters backwards, one per line.*

The task of traversing or iterating through a string is so common that Python provides an alternate, simpler, syntax: `for` loop.

```
for char in fruit:
  print char
```

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until there are no characters left.

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. Here is a loop that outputs these names in order:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for letter in prefixes:
  print letter + suffix
```

The output of this program is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because we've misspelled "Ouack" and "Quack."

As an exercise, modify the program to correct this error.

7.4 Slicing

It is common to read part of a larger string, which in Python is called a **slice**. The syntax for a slice expression is similar to the syntax for extracting a single character.

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

The operator `[n:m]` returns the part of the string from the `n`th character to the `m`th character, including the first, but excluding the last. This behavior is counterintuitive, but it might make more sense if you picture 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:

```
>>> x = 'banana'
>>> x[:3]
'ban'
>>> x[3:]
'ana'
```

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

7.5 `string` comparison

The comparison operators in Section 4.2 also work on `strings`. To see if two strings are equal:

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

Other comparison operations are useful for putting words in alphabetical order.

```
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 the Python does not handle upper and lower case letters the same way that people do. All the upper case letters come before all the lower case letters. As a result,

```
Your word, Zebra, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, like all lower-case, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

7.6 strings are not mutable

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing one of the letters. For example:

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

Instead of producing the output Jello, world!, this code produces an error message like

```
TypeError: object doesn't support item assignment
```

An alternative is to create a new string by concatenating a new character and the remainder of the original string:

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

But keep in mind that this operation does not modify the original string.

7.7 A find function

Take a look at the following function and figure out what it does.

```
def find(str, ch):
  index = 0
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

In a sense `find` 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 first appears. If the character is not found, the function returns `-1`.

It is possible for the function to return before it traverses the entire string. If `str[index] == ch`, the function returns immediately. If we get all the way through the loop without returning, then the letter must not appear in the string, and the function returns `-1`.

This traversal pattern is common, and sometimes called a eureka pattern because as soon as we find what we are looking for, we can cry "Eureka!" and stop looking.

> *As an exercise, modify the* `find` *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
index = 0
for char in fruit:
  if char == 'a':
    count = count + 1
print count
```

This program demonstrates another common idiom, called a **counter**. The variable `count` is initialized to zero and then incremented each time we find an `'a'`. (To **increment** is to increase by one; it is the opposite of **decrement**, and unrelated to excrement, which is a noun.) When we exit the loop, `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 arguments.*
>
> *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 section.*

7.9 The `string` module

The string module contains a number of functions that are useful for manipulating strings. We have to import the string module before we use the functions in it.

```
>>> import string
```

The module includes a function named `find` that does the same thing as the function we wrote. To call it,

```
>>> fruit = "banana"
>>> index = string.find(fruit, "a")
>>> print index
1
```

Again, we have to specify the name of the module and the name of the function. This example demonstrates one of the benefits of modules; they help avoid collisions between the names of built-in functions and user-defined functions. In this case we can specify which version of `find` we want using the dot operator.

Actually, `string.find` is more general than the version we wrote. First, it can find substrings, not just characters:

```
>>> string.find("banana", "na")
2
```

Also, it takes additional arguments that specify the index where it should start looking:

```
>>> string.find("banana", "na", 3)
4
```

Or the index range it should search in:

```
>>> string.find("bob", "b", 1, 2)
-1
```

In this example, the search fails because the letter `"b"` does not appear in the index range from `1` to `2` (not including `2`).

There are many other functions in the string module that we will not explain here. Once you know how to use a few of them, the rest are straightforward. You can read about them in section 4.1 of the Python Library Reference, by Guido van Rossum. That document and a wealth of other information about Python can be found at the Python website, http://www.python.org. You have now reached the stage in your study of Python where it will benefit you to begin using the website as a resource.

7.10 Character classification

It is often useful to examine a character and test whether it is upper or lower case, or whether it is a character or a digit. The string module provides several string constants that are useful for these purposes.

The string `string.lowercase` contains all the letter the system considers lower case. Similarly, `string.uppercase` contains all the upper case letters. Try the following and see what you get:

```
>>> print string.lowercase
>>> print string.uppercase
>>> print string.digits
```

We can use these constants and `find` to classify characters. For example, if `find(lowercase, ch)` returns a value other than `-1`, then `ch` must contain a lower case letter.

```
def isLower(ch):
  return find(string.lowercase, ch) != -1
```

Alternatively, we can take advantage of the `in` operator, which determines whether a character appears in a string:

```
def isLower(ch):
  return ch in string.lowercase
```

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

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

If `ch` is between `'a'` and `'z'`, it must be a lower case letter. Which of these do you think will be fastest? Can you think of another reason to prefer one or the others?

7.11 Glossary

compound data type: A data type whose values are made up of components, or elements, that are themselves values.

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

index: A variable or value used to select one of the members of an ordered set, like a character from a string.

slice:

mutable: Compound data types in Python are said to be *mutable* when their elements can be assigned new values.

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

increment: Increase the value of a variable by one.

decrement: Decrease the value of a variable by one.

concatenate: To join two operands end-to-end.

ASCII: American Standard Code for Information Interchange. A common code for storing characters in a computer.

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.

List and strings, and anything else that behaves like an ordered set, are called **sequences**. There are a number of operations that can be applied to any kind of sequence.

8.1 List values

There are several ways to create a new list; the simplest is to enclose the elements in square brackets:

```
[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 have the same type. The following list contains a string, a float, and an integer, and the last element is another list, `[10, 20]`.

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

A list within another list is said to be **nested**.

Lists that contain consecutive integers are common, so Python provides a simple way to create them:

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

The `range` function takes two arguments and returns a list that contains all the integers from the first to the second, including the first, but not including the second!

There are two alternate forms of `range`. With a single argument, it creates a list that starts at 0:

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

If there is a third argument it specifies the space between successive values, sometimes called the step. This example counts from 1 to 10 by steps of 2:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Finally, there is a special list that contains no elements. It is called the empty list and it is denoted `[]`.

With all these ways to create lists, it would be disappointing if we couldn't assign list values to variables or pass lists as parameters to functions. We can.

```
vocabulary = ["ameliorate", "castigate", "defenestrate"]
numbers = [17, 123]
empty = []
print vocabulary, numbers, empty
['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: the square bracket operator `[]`. The expression inside the square brackets specifies the index. Remember that the indices start at zero.

```
print numbers[0]
numbers[1] = 5
```

The `[]` 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 expression with type `int` can be used as an index.

```
>>> numbers[3-2]
5
>>> numbers[1.0]
TypeError: sequence index must be integer
```

If you try to access (read or write) an element that does not exist, you will get an `IndexError`.

```
>>> numbers[2] = 5
IndexError: list assignment index out of range
```

Because the indices start at zero, there is no element with the index 2. If an index has a negative value, it counts backwards starting at the end of the list.

```
>>> numbers[-1]
5
>>> numbers[-2]
17
>>> numbers[-3]
IndexError: list index out of range
```

`numbers[-1]` is the last element of the list, `numbers[-2]` is the second to last, and `numbers[-3]` doesn't exist.

One of the most common ways to index an array is with a loop variable. For example:

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

This `while` loop counts from 0 up to 4, and when the loop variable `i` is 4, the condition fails and the loop terminates. Thus, the body of the loop is only executed when `i` is 0, 1, 2 and 3.

Each time through the loop we use the variable `i` as an index into the list, printing the `i`th element. This type of **list traversal** is very common.

8.3 List length

The function `len` takes a list and returns the length of the list. It is a good idea to use this value as the upper bound of a loop, rather than 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", "pestiness", "death"]
i = 0
while i < len(horsemen):
  print horsemen[i]
  i = i + 1
```

The last time the body of the loop gets 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, since it would cause an error.

Although a list can contain another list as an element, the nested list still counts as a single element. The length of this list

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

is 4.

8.4 Lists and `for` loops

Using a loop to traverse the elements of a list is so common that Python provides a special operation for it, the `for` loop.

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

This statement is equivalent to

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

Except that is it not necessary to use the loop variable `i`. Using this syntax, the previous loop is much simpler:

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

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

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

The first example eats all the fruit in the list. The second example prints all the prime numbers between 1 and 10 (assuming that the function `is_prime` is defined somewhere). Notice that we don't bother checking the even numbers.

8.5 List operations

Lists support several additional operations. The + operator concatenates lists:

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

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

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In the first example the list `[0]` contains a single element that is repeated four times. In the second example, the list `[1, 2, 3]` is repeated three times.

The `del` statement removes an element from a list.

```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> a
['one', 'three']
```

8.6 Slices

All of the slice operations that apply to strings also work on lists.

```
>>> list = ['a', 'b', 'c', 'd', 'e']
>>> list[1:3]
['b', 'c']
>>> list[:4]
['a', 'b', 'c', 'd']
>>> list[3:]
['d', 'e']
>>> list[:]
['a', 'b', 'c', 'd', 'e']
```

In addition, slice operations on lists can appear on the left hand side of an assignment statement. This is not true with strings, because strings are immutable.

The following example replaces multiple items in a list.

```
>>> list = ['a', 'b', 'c', 'd', 'e']
>>> list[1:3] = ['x', 'y']
>>> print list
['a', 'x', 'y', 'd', 'e']
```

You can also remove elements from a list by assigning the empty list to them.

```
>>> list = ['a', 'x', 'y', 'd', 'e']
>>> list[1:3] = []
>>> list
['a', 'd', 'e']
```

And you can add elements to a list by squeezing them into a slice with only one element.

```
>>> list = ['a', 'd', 'e']
>>> list[1:1] = ['b', 'c']
>>> list
['a', 'b', 'c', 'd', 'e']
```

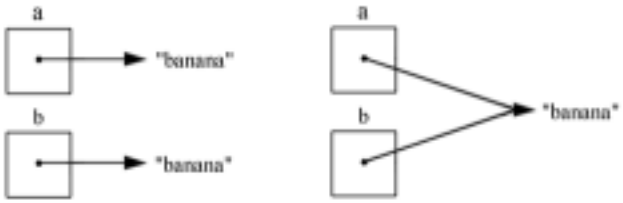
...and `list` is back where it began.

8.7 Objects and values

Consider the following assignments:

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

Clearly `a` and `b` have the same value, the string `"banana"`. But there are two possible states that might result:



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 a thing that can get referred to.

You might wonder which arrangement Python actually uses. Is there an experiment you can perform to figure out which it is?

```
>>> id(a)
135044008
>>> id(b)
135044008
```

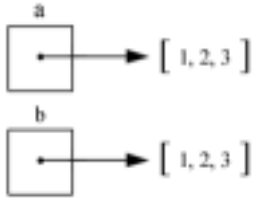
Every object has a unique identifier; the `id` function returns the unique identifier of the given object. In this case we get the same id twice, indicating that `a` and `b` refer to the same object.

Because strings are immutable, there is no practical difference between the two possible states. But for mutable types like lists, it matters.

When you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

So the state diagram looks like this:



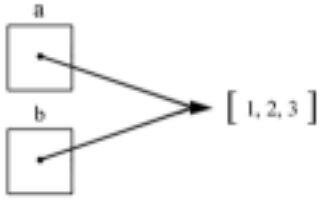
`a` and `b` have the same value, but they do no refer to the same object.

8.8 Aliasing

Variables contain references to objects. If you assign one variable to another, it means that both variables refer to the same object.

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

In this case, the state diagram looks like this:



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

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Although this behavior can be useful, it is sometimes unexpected or undesirable. For example, 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 its reference. This process is sometimes called cloning, to avoid the ambiguity of the word "copy."

8.9 Cloning lists

There is no built-in Python command to clone lists, but we can get the same effect using slices.

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

We start by initializing `b` to the empty list. Then we take a slice of `a` that consists of the whole list, and use it to replace a slice of `b`.

As an exercise, draw a state diagram of the result.

Now we are free to make changes to `b` without worrying about `a`:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

8.10 List parameters

When you pass a list as an argument, you are passing a reference to the list. For example, the function `head` takes a list as a parameter and returns the first element.

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

Here's how it is used.

```
>>> numbers = [1,2,3]
>>> head(numbers)
1
```

In this case, the parameter `list` is an alias for the variable `numbers`. If the function modifies a list passed as a parameter, the caller will see the change. `delete_head` removes the first element from a list.

```
def delete_head(list):
  del list[0]
```

Here's how `delete_head` is used.

```
>>> numbers = [1,2,3]
>>> delete_head(numbers)
>>> print numbers
[2, 3]
```

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

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

Here's how tail is used:

```
>>> numbers = [1,2,3]
>>> rest = tail(numbers)
>>> print rest
>>> [2, 3]
```

In this case the original list is unmodified. You might wonder whether changes to `rest` affect `numbers`. Try it and find out.

> *As an exercise, write a function called* `clone_list` *that takes a list as a parameter and that returns a cloned list.*

8.11 Nested lists

We have already seen an example of a nested list:

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

The three-eth element of this list is itself a list. If we print `ist[3]`, we get `[10, 20]`. To extract the elements of the nested list, we can proceed in two steps:

```
>>> elt = list[3]
>>> elt[0]
10
```

Or we can combine them:

```
>>> list[3][1]
20
```

It is common to use nested lists to represent matrices. For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

might be represented

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

That is, `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.

```
>>> matrix[1]
[4, 5, 6]
```

Or we can get a single element from the matrix using the double-index form.

```
>>> matrix[1][1]
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. In the next chapter we will see a more radical alternative using a dictionary.

8.12 Glossary

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

index: An integer variable or value used to indicate an element of a list.

sequence: Any of the data types which consist of an ordered set of elements, with each element identified by an index. The three sequence types in Python are `strings`, `lists`, and `tuples`.

element: One of the values in a list (or other sequence). The `[]` operator selects elements of a list.

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 that a variable can refer to.

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

Chapter 9

Histograms

9.1 Random numbers

Most computer programs do the same thing every time they are executed, so they are said to be **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. For some applications, though, we would like the computer to be unpredictable. Games are an obvious example, but there are many more.

Making a program truly *nondeterministic* turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to generate random numbers and use them to determine the outcome of the program. Python provides a built-in function that generates **pseudorandom** numbers, which are not truly random in the mathematical sense, but for our purposes, they will do.

The `random` module contains a function called `random` that returns a `float` between 0.0 and 1.0. Each time you call `random` you get a different randomly-generated number. To see a sample, run this loop:

```
import random

for i in range(10):
  x = random.random()
  print x
```

To generate a random `float` between 0.0 and an upper bound like `high`, you can multiply `x` by `high`.

> *As an exercise, generate a random number between* `low` *and* `high`.
>
> *As an additional exercise, generate a random integer between* `low` *and* `high`.

9.2 Statistics

The numbers generated by `random` are supposed to be distributed uniformly. If you have taken statistics, you know what that means. Among other things, it means that if we divide the range of possible values into equal sized "buckets," and count the number of times a random value falls in each bucket, each bucket should get the same number of hits (eventually).

In the next few sections, we will write programs that generate a sequence of random numbers and check whether this property holds true.

9.3 List of random numbers

The first step is to generate a large number of random values and store them in a list. By "large number," of course, we mean 8. It's always a good idea to start with a manageable number, to help with debugging, and then increase it later.

The following function takes a single argument, the size of the list. It creates a new list of 0s and then replaces the elements with random values. The return value is a reference to the new list.

```
def random_list(n):
  s = [0]*n
  for i in range(n):
    s[i] = random()
  return s
```

To test this function, it is convenient to have a function that prints the list one element per line.

```
def print_list(s):
  for elt in s:
    print elt
```

The following code generates a list and prints it:

```
num_values = 8
s = random_list(num_values)
print_list(s)
```

When we ran this code, the output was

```
0.15156642489
0.498048560109
0.810894847068
0.360371157682
0.275119183077
0.328578797631
0.759199803101
0.800367163582
```

which is pretty random-looking. Your results may differ.

If these numbers are really random, we expect half of them to be greater than 0.5 and half to be less. In fact, three are greater than 0.5, so that's a little low.

If we divide the range into four buckets–from 0.0 to 0.25, 0.25 to 0.5, 0.5 to 0.75, and 0.75 to 1.0–we expect 2 values to fall in each bucket. In fact, we get 1, 4, 0, 3. Again, not exactly what we expected.

Do these results mean the values are not really random? It's hard to tell. With so few values, the chances are slim that we would get exactly what we expect. But as the number of values increases, the outcome should be more predictable.

To test this theory, we'll write some programs that divide the range into buckets and count the number of values in each.

9.4 Counting

A good approach to problems like this is to think of simple functions that are easy to write, and that might turn out to be useful. Then you can combine them into a solution. Of course, it is not easy to know ahead of time which functions are likely to be useful, but as you gain experience you will have a better idea.

Also, it is not always obvious what sort of things are easy to write, but a good approach is to look for subproblems that fit a pattern you have seen before.

Back in Section 7.7 we looked at a loop that traversed a string and counted the number of times a given letter appeared. You can think of this program as an example of a pattern called "traverse and count." The elements of this pattern are:

- A sequence of elements, like a list or a string, that can be traversed.
- A test that you can apply to each element in the sequence.
- A counter that keeps track of how many elements pass the test.

In this case, we have a function in mind called `in_bucket` that counts the number of elements in a list that fall in a given bucket. The arguments are the list and two numbers that specify the lower and upper bounds of the bucket.

```
def in_bucket(list, low, high):
  count = 0
  for elt in list:
    if low <= elt < high:
      count = count + 1
  return count
```

We haven't been very careful about whether something equal to `low` or `high` falls in the bucket, but you can see from the code that `low` is in and `high` is out. That should prevent us from counting any elements in more than one bucket.

Now, to divide the range into two pieces, we could write

```
low = in_bucket(a, 0.0, 0.5)
high = in_bucket(a, 0.5, 1)
```

To divide it into four pieces:

```
bucket1 = in_bucket(a, 0.0, 0.25)
bucket2 = in_bucket(a, 0.25, 0.5)
bucket3 = in_bucket(a, 0.5, 0.75)
bucket4 = in_bucket(a, 0.75, 1.0)
```

You might want to try out this program using a larger `num_values`. As `num_values` increases, are the numbers in each bucket leveling off?

9.5 Many buckets

Of course, as the number of buckets increases, we don't want to have to rewrite the program, especially since the code is getting big and repetitive. Any time you find yourself doing something more than a few times, you should be looking for a way to automate it.

Let's say that we wanted 8 buckets. The width of each bucket would be one eighth of the range, which is 0.125. To count the number of values in each bucket, we need to be able to generate the bounds of each bucket automatically, and we need to have some place to store the 8 counts.

We can solve the first problem with a loop:

```
num_buckets = 8
bucket_width = 1.0 / num_buckets
for i in range(num_buckets):
  low = i * bucket_width
  high = low + bucket_width
  print low, "to", high
```

This code uses the loop variable `i` to multiply by the bucket width, in order to find the low end of each bucket. The output of this loop is:

```
0.0 to 0.125
0.125 to 0.25
0.25 to 0.375
0.375 to 0.5
0.5 to 0.625
0.625 to 0.75
0.75 to 0.875
0.875 to 1.0
```

You can confirm that each bucket is the same width, that they don't overlap, and that they cover the whole range from 0.0 to 1.0.

Now we just need a way to store 8 integers, preferably so we can use an index to access each one. Immediately, you should be thinking "list!"

We have to create the list outside the loop (because we only want to do it once). Inside the loop we'll call `in_bucket` repeatedly and put the results in the list:

```
list = random_list(1000)

num_buckets = 8
buckets = [0] * num_buckets
bucket_width = 1.0 / num_buckets
for i in range(num_buckets):
  low = i * bucket_width
  high = low + bucket_width
  #print low, "to", high
  buckets[i] = in_bucket(list, low, high)
print buckets
```

This code works. We cranked the number of values up to 1000 and divided the range into 8 buckets. The output was:

```
[138, 124, 128, 118, 130, 117, 114, 131]
```

which is pretty close to 125 in each bucket. At least, it's close enough that we can believe the random number generator is working.

9.6 A single-pass solution

Although this code works, it is not as efficient as it could be. Every time it calls `in_bucket`, it traverses the entire list. As the number of buckets increases, that gets to be a lot of traversals.

It would be better to make a single pass through the list and computer, for each value, the index of the bucket it falls in. Then we could increment the appropriate counter.

In the previous section, we took an index, `i`, and multiplied it by the `bucketWidth` in order to find the lower bound of a given bucket. Now we want to take a value in the range 0.0 to 1.0, and find the index of the bucket where it falls.

Since this problem is the inverse of the previous problem we might guess that we should *divide* by the `bucket_width` instead of multiplying. That guess is correct.

Remember that since `bucket_width = 1.0 / num_buckets`, dividing by `bucket_width` is the same as multiplying by `num_buckets`. If we take a number in the range 0.0 to 1.0 and multiply by `num_buckets`, we get a number in the range from 0.0 to `numBuckets`. If we round that number to the next lower integer, we get exactly what we are looking for–the index of the appropriate bucket.

```
list = random_list(1000)

num_buckets = 8
buckets = [0] * num_buckets
for i in list:
  index = int(i * num_buckets)
  buckets[index] = buckets[index] + 1
```

Here we are using the `int` function to convert a floating point number to an integer.

Is it possible for this calculation to produce an index that is out of range (either negative or greater than `len(buckets)-1`)?

A list like `buckets`, that contains counts of the number of values in each range, is called a **histogram**.

> *As an exercise, write a function called* `histogram` *that takes an array and a number of buckets as parameters, and that returns a histogram with the given number of buckets.*

9.7 Glossary

deterministic: A program that does the same thing every time it is called.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

histogram: A list of integers where each integer counts the number of values that fall into a certain range.

Chapter 10

Tuples and dictionaries

10.1 Mutability and tuples

We have seen two compound types: strings, which are made up of characters, and lists, which are made up of elements of any type. One of the differences we noted is that you can modify the elements of a list but you cannot modify the characters in a string. In other words, strings are **immutable** and lists are **mutable**.

There is another type in Python, called a **tuple**, that is similar to a list except that it is immutable. Syntactically, a tuple is a comma-separated list of values:

```
>>> mytuple = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is conventional to enclose tuples in parentheses.

```
>>> mytuple = ('a', 'b', 'c', 'd', 'e')
```

If you create a tuple with a a single element, you *must* include a final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Without the comma, Python treats `('a')` as a string in parentheses.

```
>>> t2 = ('a')
>>> type(t2)
<type 'string'>
```

Syntax aside, the operations on tuples are the same as the operations on lists:

```
>>> mylist = ['a', 'b', 'c', 'd', 'e']
>>> mylist[0]               #index
'a'
>>> mytuple[0]
'a',
>>> mylist[1:3]             #slice
['b', 'c']
>>> mytuple[1:3]
('b', 'c')
```

But if we try to modify one of the elements of the tuple we get an error:

```
>>> mylist[1] = 'A'         # assign 'A' to the first element of list
>>> mytuple[1] = 'A'      # try to assign 'A' to the 1st element of tuple
TypeError: object doesn't support item assignment
```

Of course, even if we can't modify the elements of a tuple, we can always replace a tuple with a different tuple:

```
>>> mytuple = ('A',) + mytuple[1:]
>>> mytuple
('A', 'b', 'c', 'd', 'e')
```

10.2 Multiple assignment

Once in a while it is useful to swap the values of two variables. To do this with conventional assignment statements, we have to use a temporary variable. For example, to swap `a` and `b`:

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

If you have to do things like this often, this approach is cumbersome. Python provides a form of **multiple assignment** that solves this problem neatly:

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

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. The interesting thing is that the assignments are performed simultaneously, not one at a time. This feature makes multiple assignment quite versatile.

The number of variables and the number of values have to be the same:

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

10.3 Tuples as return values

Function can return tuples as return values. For example, we could write a function that takes two parameters and swaps them:

```
def swap(x, y):
  return y, x
```

When we call this function we have to assign the return value to a tuple with two variables.

```
a, b = swap(a, b)
```

In this case there is no great advantage in making `swap` a function. In fact, there is a danger in trying to encapsulate `swap`, which is the following tempting mistake:

```
def swap(x, y):      # incorrect version
  x, y = y, x
```

If we call this function like this,

```
swap(a, b)
```

then `a` and `x` are aliases for the same value. Changing `x` inside `swap` makes `x` refer to a different value, but it has no effect on `a` in `__main__`. Similarly, changing `y` has no effect on `b`.

> *As an exercise, draw a state diagram for this function so that you can see why it does not work as desired.*

10.4 Dictionaries

The compound types we have looked at—strings, lists and tuples—use integers as indices. If you try to use any other type as an index you get an error.

Dictionaries are similar to other compound types except that they can use any immutable type as an index. As an example, we will create a dictionary to translate English words into Spanish. For this dictionary, the indices are `strings`.

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

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

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

```
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```

The elements of a dictionary appear in a comma-separated list. Each entry contains an index and a value separated by a colon. In a dictionary the indices are called **keys**, so the elements are called **key-value pairs**.

Another way to create a dictionary is to provide a list of key-value pairs using the same syntax as the previous output:

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

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

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

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

```
>>> print eng2sp['two']
'dos'
```

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

10.5 Dictionary operations

The `del` statement removes a key-value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock:

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

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

```
>>> del inventory['pears']
>>> print inventory
{'oranges': 525, 'apples': 430, 'bananas': 312}
```

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

```
>>> inventory['pears'] = 0
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}
```

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

```
>>> len(inventory)
4
```

10.6 Dictionary methods

A **method** is similar to a function—it takes parameters and returns a value—but the syntax is different. For example, the `keys` method takes a dictionary and returns a list of the keys that appear, but instead of the function syntax `keys(eng2sp)`, we use the method syntax `eng2sp.keys()`.

The dot operator specifies the name of the function, `keys`, and the name of the object to apply the function to, `eng2sp`. The parentheses indicate that this method takes no arguments. A method call is called an **invocation**; in this case we would say that we are invoking `keys` on the object `eng2sp`.

The `values` method is similar; it returns a list of the items in the dictionary:

```
>>> eng2sp.values ()
['uno', 'tres', 'dos']
```

If a method takes an argument, it uses the same syntax as a function call. For example, the method `has_key` takes a key and returns true if the key appears is in the dictionary and false if it doesn't.

```
>>> eng2sp.has_key('one')
1
>>> eng2sp.has_key('deux')
0
```

If you try to invoke a method without specifying an object, you get an error. In this case the error message is not very helpful:

```
>>> has_key('one')
NameError: has_key
```

10.7 Aliasing and copying

Because dictionaries are mutable, we need to be aware of aliasing. If two variables refer to the same object, they are aliases; any changes to one affect the other.

If you want to modify a dictionary and keep a copy of the original, you can use the `copy` method. For example, `opposites` is a dictionary that contains pairs of opposites:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

`alias` and `opposites` refer to the same object; `copy` refers to a fresh copy of the same dictionary. If we modify `alias`, `opposites` sees the change:

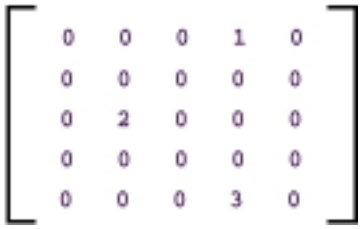
```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

If we modify `copy`, `opposites` is unchanged:

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

10.8 Sparse matrices

In section 8.11 we used a list of lists to represent a matrix. For a matrix with mostly non-zero values, that is a good choice, but consider a sparse matrix like



The list representation contains a lot of zeroes:

```
matrix = [[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 dictionary along with the `get` method. For the keys we can use tuples that contains the row and column numbers. Here is the dictionary representation of the same matrix:

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

There are only three key-value pairs, one for each non-zero element of the matrix. Each key is a tuple and each value is an integer. The reason we used tuples as keys, rather than lists, is that the keys have to be immutable.

To access an element of the matrix, we could use the `[]` operator:

```
matrix[0,3]
1
```

The only problem is that if we specify an element that is zero, we get an error, because there is no entry in the dictionary with that key.

```
>>> matrix[1,3]
KeyError: (1, 3)
```

The `get` method solves this problem.

```
>>> matrix.get((0,3), 0)
1
```

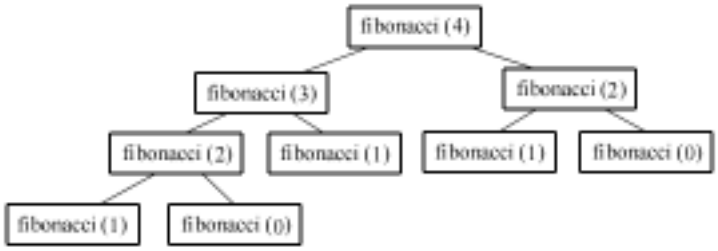
The first argument is the key; the second argument is the value we would like `get` to return if the key is not in the dictionary.

```
>>> matrix.get((1,3), 0)
0
```

10.9 Hints

If you played around with the `fibonacci` function from Section 5.8, 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 our machine, `fibonacci(20)` finishes instantly, `fibonacci(30)` takes about a second, and `fibonacci(40)` takes roughly forever.

To understand why it takes so long, consider this call graph for `fibonacci(4)`.



The call graph shows each function and the function calls it makes. At the top of the graph, `fibonacci(4)` calls `fibonacci(3)` and `fibonacci(2)`. In turn, `fibonacci(3)` calls `fibonacci(2)` and `fibonacci(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 dictionary. A previously computed value that is stored for later use is called a **hint**, and the following code makes extensive use of them:

```
previous = {0:1, 1:1}
def fibonacci(n):
  if previous.has_key(n):
    return previous[n]
  else:
    new_value = fib(n-1) + fib(n-2)
    previous[n] = new_value
    return new_value
```

The dictionary named `previous` keeps track of the Fibonacci numbers we already know. At the beginning of the program we start with only two pairs: 0 maps to 1 and 1 maps to 1.

Any time `fibonacci` is called, it checks the dictionary to see if it contains a precomputed result, sometimes called a hint or a memo. If it's there, we can return the value immediately without making any more recursive calls. If not, we have to compute the new value. Once we get it, we add it to the dictionary.

Using this version of `fibonacci`, my machine can compute `fibonacci(40)` in an eyeblink. But when I try to compute `fibonacci(50)`, I get a different problem

```
>>> fibonacci(50)
OverflowError: integer addition
```

The answer, as we'll see in a minute, is 20,365,011,074. The problem is that this number is too big to fit into a Python integer. It "overflows". Fortunately, there is an easy solution for this problem.

10.10 Long integers

Python provides a type called `long int` that can handle any size integer. There are two ways to create a `long int` value. One is to write an integer with a capital L at the end.

```
>>> type(1L)
<type 'long int'>
```

The other is to use the `long` function to convert a value to a `long int`. `long` can accept any numerical type, and even strings of digits:

```
>>> long(1)
1L
>>> long(3.9)
3L
>>> long('57')
57L
```

All the math operations work on `long ints`, so we don't have to do much to make `fibonacci` generate `long ints`.

```
>>> previous = {0:1L, 1:1L}
>>> fibonacci(50)
20365011074L
```

Just by changing the initial contents of `previous`, we change the behavior of `fibonacci`. The first two numbers in the sequence are `long ints`, so all the subsequent numbers in the sequence are, too.

> *As an exercise, convert* `factorial` *so that it produces a* `long int` *as a result.*

10.11 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, for every letter we would like to know how many times it appears.

One reason such a histogram might be useful is for compressing a text file. Because different letters appear with different frequencies, we can compress a file by using short codes for common letters and longer codes for less frequent letters.

Dictionaries provide an elegant way to generate a histogram:

```
>>> letterCounts = {}
>>> for letter in "Mississippi":
...   letterCounts[letter] = letterCounts.get (letter, 0) + 1
...
>>> letterCounts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
>>>
```

Initially we have an empty dictionary. For each letter in the string, we find the current count (possibly zero) and increment it. At the end, the dictionary contains pairs of letters and their frequencies.

It might more appealing to display the histogram in alphabetical order. We can use the `sort` method.

```
>>> letterItems = letterCounts.items()
>>> letterItems.sort()
>>> print letterItems
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
>>>
```

We have seen the `items` method before, but `sort` is the first method we have seen that applies to lists. There are several other list methods, including `append`, `extend` and `reverse`. You should consult the Python documentation for more details.

10.12 Glossary

mutable type: A data type whose elements can be modified. All mutable types are *compound types*. Simple types such as integers and floats are not mutable. Lists and dictionaries are mutable data types, strings and tuples are not.

immutable type: A type whose elements can not be modified. Assignments to elements or slices of immutable types will result in an error.

tuple: A sequence type that is similar to a list except that it is immutable. Tuples can be used wherever an immutable type is required, such as a key in a dictionary.

multiple assignment: Assignment to all the elements in a tuple using a single assignment statement. Multiple assignment occures in parallel rather than in sequence, making it useful for swapping values.

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

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

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

invoke: To call a method.

Chapter 11

Classes and objects

11.1 User-defined compound types

Having used some of Python'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 we treat 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 Python is with two floating point values. The question, then, is how to group these two values into a compound object. The answer is to define a new user-defined compound type, which is called a **class**.

Here's what a class definition looks like.

```
class Point:
  pass
```

`class` definitions can appear anywhere in a program, but they are usually near the beginning (after the `import` statements). The syntax for creating a `class` definition is another example of the compound statements that we discussed in section 4.3.

This definition simply creates a new type called `Point`. The second line is a **pass** statement that has no effect; it is only necessary because a compound statement must have something in its body.

Next we would like to create an object with type `Point`. Creating a new member of a class is called **instantiation** and the new **object** is called an **instance** of the class. To create a new instance:

```
blank = Point()
```

The variable `blank` gets assigned a reference to a new `Point` object. The parentheses indicate that `Point` is the name of a function as well as the name of the class. A function like `Point` that creates new objects is called a **constructor**.

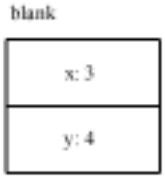
11.2 Instance variables

We can add new components to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, like `math.pi` or `string.uppercase`. In this case, though, we are selecting a component from an instance. These components are called **instance variables**.

The following state diagram shows the result of these assignments:



The value of `blank` is a reference to the new object, which contains two instance variables. Each instance variable refers to a floating-point number.

We can read the values of an instance variable using the same syntax:

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

The expression `blank.x` means "go to the object `blank` refers to and get the value of `x`." In this case we assign that value to a local variable named `x`. There is no conflict between the local variable `x` and the instance variable `x`. 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 are legal.

```
print '(' + str(blank.x) + ', ' + str(blank.y) + ')'
distance = blank.x * blank.x + blank.y * blank.y
```

The first line outputs `(3.0, 4.0)`; the second line calculates the value 25.

Finally, you might be tempted to print the value of `blank` itself.

```
>>> print blank
<__main__.Point instance at 80f8e70>
```

This indicates that `blank` is an instance of the `Point` class and it was defined in `__main__`. `80f8e70` is the unique identifier for this object in hexadecimal.

> *As an exercise, translate it into decimal and compare it with the result of the* `id` *function.*

11.3 Instances as parameters

You can pass an instance as a parameter in the usual way. For example:

```
def printPoint(p):
  print '(' + str(p.x) + ', ' + str(p.y) + ')'
```

`printPoint` takes a point as an argument and outputs it in the standard format. If you call `printPoint(blank)`, it will output `(3, 4)`.

> *As an exercise, rewrite the* `distance` *function from Section 5.2 so that it takes two* `Points` *as parameters instead of four numeric values.*

11.4 Rectangles

Now let's say that we want to create 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 let's assume that the rectangle will be oriented 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:
  pass
```

And instantiate it:

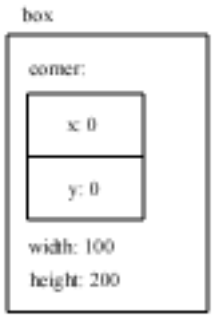
```
box = Rectangle()
box.width = 100.0
box.height = 200.0
```

This code creates a new `Rectangle` object and two floating-point instance variables. To specify the upper left corner we can create an object within an object!

```
box.corner = Point()
box.corner.x = 0.0;
box.corner.y = 0.0;
```

The dot operator composes. The expression `box.corner.x` means "go to the object `box` refers to and select the component named `corner`; then go to that object and select the component named `x`."

The figure shows the state of this object.



11.5 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):
  p = Point()
  p.x = box.corner.x + box.width/2.0
  p.y = box.corner.y + box.height/2.0
  return p
```

To call this function, we pass a box as an argument and assign the result to a variable:

```
>>> center = findCenter (box)
>>> printPoint (center)
```

The output of this program is `(50, 100)`.

11.6 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.

pass: A Python statement which essentially says "do nothing". It is used in the body of compound statements when no action is desired but syntax rules require that a statement be present.

instantiate: To create an instance of a class.

instance: An object that belongs to a class.

object: "Object" is one of those fundemental terms that are difficult to define. Intuitively, an object models things (objects) in the real world. It has form and it has behavior. The form in a Python object consists in the data values that are part of the object. The instance variables discussed in this chapter are examples of this. The behavior of a Python object is determined by the functions or methods which are part of that object. We will first look at methods in Chapter 13. All instances of classes in Python are objects, but not all objects are instances of classes. Python's built-in types are not classes, yet all values with these types are properly called objects.

constructor: A method which is automatically invoked when an object of that class is instantiated.

instance variable: One of the named data items that make up an instance.

Chapter 12

Classes and functions

12.1 Time

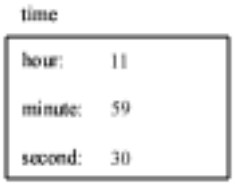
As a second example of a user-defined object, we will define a class called `Time` that records the time of day. Again, the class definition looks like this:

```
class Time:
  pass
```

We can now create a new `Time` object as we did before, and create instance variables to contain an hours, minutes and seconds:

```
time = Time()
time.hours = 11
time.minutes = 59
time.seconds = 30
```

The state diagram for this object looks like this:



As an exercise, write a function `printTime` *that takes a Time object as an argument and prints it in the form:* `hours:minutes:seconds`.

As a second exercise, write a boolean function `after` *that takes two Times,* `t1` *and* `t2` *as arguments and returns true (1) if* `t1` *follows* `t2` *chronologically, and false (0) otherwise.*

12.2 Pure functions

In the next few sections we will write several versions of a function called `addTime` that calculates the sum of two Times. They serve as examples of two kinds of functions: pure functions and modifiers. Also, the approach we take to writing these functions demonstrates once again the method of programming called incremental development.

In incremental development, we often start with a rough draft of a function that is syntactically correct, and that does something almost right, but that is not complete. Here is a rough version of `addTime`:

```
def addTime(t1, t2):
  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 instance variables, and returns a reference to the new object. We would say that this is a **pure function** because it does not modify any of the objects passed to it as parameters, and it has no side-effects like printing something 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 breadmaker to make bread. Then we'll use `addTime` to figure out when the bread will be done.

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds =  30

>>> breadTime = Time()
>>> breadTime.hours =  3
>>> breadTime.minutes =  35
>>> breadTime.seconds =  0

>>> doneTime = addTime(currentTime, breadTime)
>>> printTime(doneTime)
```

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 60. When that happens we have to "carry" the extra seconds into the minutes column, or extra minutes into the hours column.

Here's a second, corrected version of this function.

```
def addTime(t1, t2):
  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 it's correct, it's starting to get big. A little later, we will suggest an alternate approach to this problem that will be much shorter.

12.3 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**.

A useful function that would be written most naturally as a modifier is `increment`, which adds a given number of seconds to a `Time` object. Again, a rough draft of this function looks like:

```
def increment(time, seconds):
  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 argument `secs` is much greater than 60? In that case, it is not enough to subtract 60 once; we have to keep doing it until `seconds` is below 60. We can do that by replacing the `if` statements with `while` statements:

```
def increment(time, secs):
  time.seconds = time.seconds + secs

  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, although it is not the most efficient solution.

As an exercise, rewrite it 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.*

12.4 Which is better?

Anything that can be done with modifiers can also be done with pure functions. In fact, there are programming languages that only allow pure functions. Some programmers believe that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, there are times when modifiers are convenient, and cases where 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**.

12.5 Incremental development versus planning

In this chapter we have demonstrated an approach to program development we refer to as **incremental 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 high-level planning, in which a little insight into the problem can make the programming much easier. In this case the insight is that a `Time` is really a three-digit number in base 60! The `second` component is the "ones column," the `minute` component is the "60's column", and the `hour` component is the "3600's 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.

Thus an alternate approach to the whole problem is to convert each `Time` object into a single integer and take advantage of the fact that the computer already knows how to do arithmetic with `int`s. Here is a function that converts a `Time` into an `int`.

```
def convertToSeconds(t):
  minutes = t.hours * 60 + t.minutes
  seconds = minutes * 60 + t.seconds
  return seconds
```

Now all we need is a way to convert from a `int` to a `Time`.

```
def makeTime(secs):
  time = Time()
  time.hours = int(secs / 3600)
  secs = secs - time.hours * 3600
  time.minutes = int(secs / 60)
  secs = secs - time.minutes * 60
  time.seconds = secs
  return time
```

You might have to think a bit to convince yourself that the technique we are using to convert from one base to another is correct. Assuming you are convinced, we can use these functions to rewrite `addTime`:

```
def addTime(t1, t2):
  seconds = convertToSeconds(t1) + convertToSeconds(t2)
  return makeTime(seconds)
```

This is much shorter than the original version, and it is much easier to demonstrate that it is correct (assuming, as usual, that the functions it calls are correct).

As an exercise, rewrite `increment` *the same way.*

12.6 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 naive 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 (more general) makes it easier (fewer special cases, fewer opportunities for error).

12.7 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 in Chapter 1, 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 most difficult 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.

12.8 Glossary

pure function: A function that does not modify any of the objects it receives as parameters. Most pure functions return a result.

modifier: A function that changes one or more of the objects it receives as parameters. Most modifiers do not have return values.

functional programming style: A style of program design in which the majority of functions are pure.

incremental development: A way of developing programs starting with a prototype and gradually testing and improving it.

algorithm: A set of instructions for solving a class of problems by a mechanical, unintelligent process.

Chapter 13

Methods

13.1 Object-oriented features

Python 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, where most of the functions operate on specific kinds of 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 we defined last chapter corresponds to the way people record the time of day, and the operations we defined correspond to the sorts of things people do with times. Similarly, the `Point` and `Rectangle` classes correspond to the mathematical concept of a point and a rectangle.

So far we have not taken advantage of the features Python provides to support object-oriented programming. Strictly speaking, these features are not necessary. For the most part they provide an alternate syntax for doing things we have already done, but in many cases the alternate syntax is more concise and more accurately conveys the structure of the program.

For example, 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, like `keys` and `values`, which we invoked on dictionaries. Each method is associated with a class and is intended to be invoked on instances of that class.

Defining a new method is similar to defining a function, with two differences:

- The method is defined inside the `class` definition, in order to make the relationship between the class and the method explicit.
- The first argument of a method is called **self**, which is a Python keyword that refers to the object on which the method is invoked.

In the next few sections, we will take the functions from the last two chapters and transform them into methods. One thing you should realize is that this transformation is purely mechanical; you can do it just 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.

13.2 printTime

In the last chapter we defined a class named `Time` and a function named `printTime`:

```
class Time:
  pass

def printTime(time):
  print str(time.hours) + ":" + str(time.minutes) + ":" + str(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)
```

To rewrite `printTime` as a method, we move the definition into the class definition and change the name of the parameter to `self`.

```
class Time:
  def printTime(self):
    print str(self.hours) + ":" + str(self.minutes) + ":" + str(self.seconds)
```

To invoke the new version of `printTime`, we invoke it on a `Time` object:

```
>>> currentTime.printTime()
```

Notice that the parameter `self` is not in the argument list when the method is invoked on the `Time` object. Instead, it is replaced implicitly with the object on which the method is invoked. It is also possible to call the `printTime` method with an explicit argument:

```
>>> Time.printTime(currentTime)
```

Try each of these and you will see that they do the same thing.

13.3 Another example

Let's convert `increment` to a method. To save space we will leave out previously defined functions, but you should keep them in your own version of the class definition.

```
class Time:
  #previous method definitions here...

  def increment(self, secs):
    secs = secs + self.seconds

    self.hours = self.hours - int(secs / 3600)
    secs = secs - self.hours * 3600
    self.minutes = int(secs / 60)
    secs = secs - self.minutes * 60
    self.seconds = secs
```

Again, the transformation is purely mechanical: we move the method definition into the class definition and change the name of the first parameter.

To invoke `increment` as a method:

```
currentTime.increment(500)
```

As an exercise, convert `convertToSeconds` *to a method in the* `Time` *class.*

13.4 A more complicated example

The `after` function is slightly more complicated because it operates on two `Time` objects, not just one. We can only convert one of the parameters to `self`; the other stays the same.

```
class Time:
  #previous method definitions here...

  def after(self, time2):
    if self.hour > time2.hour:
      return 1
    if self.hour < time2.hour:
      return 0

    if self.minute > time2.minute:
      return 1
    if self.minute < time2.minute:
      return 0
```

```
    if self.second > time2.second:
      return 1
    return 0
```

We invoke this method on one object and pass the other as an argument:

```
if doneTime.after(currentTime):
  print "The bread will be done after it starts."
```

You can almost read the invocation like English: "If the done-time is after the current-time, then..."

13.5 Optional arguments

We've seen a number of built-in functions that take a variable number of arguments. For example, `string.find` can take 2, 3 or 4 arguments.

It is also possible to write user-defined functions with optional argument lists. For example, we can upgrade our own version of `find` to do the same thing as `string.find`.

Here is the original version:

```
def find(str, ch):
  index = 0
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

Here is a new and improved version:

```
def find(str, ch, start=0):
  index = start
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

The third parameter, `start`, is optional because we have provided a default value, `0`. If we invoke `find` with only two arguments, we use the default value and start from the beginning of the string.

```
>>> find("apple", "p")
1
```

If we provide a third parameter, it overrides the default:

```
>>> find("apple", "p", 2)
2
>>> find("apple", "p", 3)
-1
```

As an exercise, add a fourth parameter, `end`*, which specifies where we should 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 we can't find it's length.*

13.6 The initialization method

The **initialization method**, also called the *constructor*, is a special method that is invoked when an object is created. The name of this method is `__init__` (two underscore characters, followed by `init` and then two more underscores). An initialization method for the time class look like this:

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

There is no conflict between the instance variable `self.hours` and the parameter `hours`. Again, the dot operator specifies which variable we are referring to.

When we invoke the `Time` constructor, we can provide arguments that are passed along to `init`.

```
>>> currentTime = Time(9, 14, 30)
>>> currentTime.printTime()
>>> 9:14:30
```

Because the parameters are optional, we can omit them:

```
>>> currentTime = Time()
>>> currentTime.printTime()
>>> 0:0:0
```

or provide only the first

```
>>> currentTime = Time (9)
>>> currentTime.printTime()
>>> 9:0:0
```

or the first two.

```
>>> currentTime = Time (9, 14)
>>> currentTime.printTime()
>>> 9:14:0
```

As with other methods, it is possible to provide a subset of the parameters by naming them explicitly:

```
>>> currentTime = Time(seconds = 30, hours = 9)
>>> currentTime.printTime()
>>> 9:0:30
```

13.7 Glossary

method: A function that is defined inside a class definition and which is invoked on instances of that class.

self: A keyword that refers to the object on which a method is invoked.

initialization method: A special method that is invoked when a new object is created.

Chapter 14

The case study

For the remainder of the book, we will take an in-depth look at a single program. Through this **case study**, called `horsebet.py`, we will look at proceedural and object-oriented design in greater detail. We will also learn about additional tools available to the Python programmer.

14.1 horsebet.py

```
# Programmers: Courtney Gleason and Katherine Smith
# Title:       horsebet.py
# Description: Race Track Game

from random import randint
import os


### Constants ###

# Horse Info:
horses = {}
horses["Thunder"] = """Thunder is a purebred.  He is one and a half years old.
Both his parents were Kentuckey Durby winners."""
horses["Mystic"] = """Mystic is a hybrid.  She is two years old.
Her mother won the Melbourne Cup at the Victoria Racing Club in Ireland.
Her father never won anything."""
horses["Bullet"] = """Bullet is a purebred.  He is three years old.
His mother was a three time national champion for jumping.
Her father won international competitions."""
horses["Golden Stallion"] = """Golden Stallion is a hybrid. He is seven years old.
Both his parents were wild horses."""
```

```
horses["Morgan"] = """Morgan is a mixed breed.  She is fifteen years old.
Her mother is a pony. Her father is a stallion."""


### Functions ###

def display_horses():
  "Print out horse roster."
  index = 1
  for horse in horses.keys():
    print str(index) + ". " + horse
    index = index + 1


def display_horse_information():
  "Displays information about horses."
  while 1:
    os.system("clear")
    print
    display_horses()
    print

    choice = raw_input("Horse name ('q' to exit): ")

    if choice == 'q': break

    if choice in horses.keys():
      print
      print "Horse: " + choice
      print
      print horses[choice]
      print
    else:
      print
      print "Sorry, horse not found."
      print

    raw_input("press <enter> to continue...")


def get_option():
  "Print options list and get choice."

  print
  print " Options: "
  print " 1. Information about horses."
```

```
  print " 2. Start the race."
  print " 3. Reset balance."
  print " 4. Quit."
  print

  choice = input("Choose (1-4): ")
  return choice


def display_horse_status(horse_status, track_length):
  "Print out the status of the race."
  os.system("clear")

  print '/' + '-'*track_length + '\\'

  for horse in horse_status.keys():
    # multiply '#' by how far the horse has advanced
    done = '#' * horse_status[horse]

    # figure out how much is left and multiply ' ' by that number
    left = ' ' * (track_length - horse_status[horse])

    # add 'em up to make a progress bar
    progress_bar = done + left

    # also append the horses name to each progress bar
    line = '|' + progress_bar + '| ' + horse
    print line

  print '\\' + '-'*track_length + '/'


def get_bet(balance):
  "Show balance and get bet."
  while 1:
    print
    print "Your current balance is: $" + str(balance)
    print
    money = input("How much would you like to bet: $")

    if money > balance:
      print "You don't have that much to spend.  Try a different amount . . .  "
    elif money < 1:
      print "Please bet more than that . . . "
    else:
      return money
```

```
def get_horse():
  "Return horse name."
  while 1:
    print "Okay, now which horse do you think is going to get first place?"
    display_horses()
    print

    choice = raw_input("Horse name: ")

    if choice in horses.keys():
      return choice
    else:
      print "Sorry, horse not found."


def get_winner(horse_status, track_length):
  "Figure out the finishline horses, and choose one for winner."

  # make a list of all the horses at the finishline
  winners = []
  for horse in horse_status.keys():
    if horse_status[horse] == track_length:
      winners.append(horse)

  # we only want ONE winner to make life easier ;-)
  # randomly slice the winners list for a single winner
  num_of_winners = len(winners)
  random_num = randint(0, num_of_winners - 1)
  winner = winners[random_num]

  return winner


def race():
  "Do the race simulation."
  track_length = 25
  horse_status = {} # status for each horse during race

  #setup horse status dictionary
  for horse in horses.keys():
    horse_status[horse] = 0 #each horse starts at 0

  #start the race!
  leading_horse = 0
```

```
  raw_input("press <enter> to begin...")
  display_horse_status(horse_status, track_length)
  # keep'em racing untill one reaches the end of the track
  while leading_horse < track_length:
    for horse in horse_status.keys():
      current_location = horse_status[horse]

      # you can make this random number thing more sophisticated by assigning
      # certain properties to each horse and have it # somehow affect the number
      # generated; as a result affecting the horses performance.
      moved = randint(0,1)

      new_location = current_location + moved

      # the horse has officialy made it's move
      horse_status[horse] = new_location
      # keep track of leading horse
      if horse_status[horse] > leading_horse:
        leading_horse = horse_status[horse] # so we know when to stop the race

    raw_input("press <enter> to continue...")
    display_horse_status(horse_status, track_length)

  winner = get_winner(horse_status, track_length)
  return winner


def simulate_race(balance):
  "The main race function."
  os.system("clear")

  bet_amnt  = get_bet(balance)
  bet_horse = get_horse()
  race_result = race()

  if bet_horse == race_result:
    print "Your horse won the race!"
    print "Horray for " + race_result + "!"
    balance = balance + bet_amnt * (len(horses) - 1)
  else:
    print "Sorry, you lose."
    print race_result + " is the winner."
    balance = balance - bet_amnt

  print "Your new balance is: " + str(balance)
```

```
  raw_input("press <enter> to continue...")

  return balance


def main():
  os.system("clear")

  print
  print "   Welcome To A Day At The Races"
  print "---------------------------------------"
  print "by Courtney Gleason & Katherine Smith"
  print

  balance = 1000

  while 1:
    choice = get_option()
    os.system("clear")
    if choice == 1:
      display_horse_information()
    elif choice == 2:
      balance = simulate_race(balance)
    elif choice == 3:
      balance = 1000
    elif choice == 4:
      break
    os.system("clear")

main()
```

14.2 Glossary

case study:

Appendix A

GNU Free Documentation License

Version 1.1, March 2000

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 noncommercially. Secondarily, 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.

A.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".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo 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.

A.2 Verbatim Copying

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.

A.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.

A.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 "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements 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.

A.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."

A.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.

A.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.

A.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.

A.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.

A.10 Future Revesions of This Licence

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.

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 © 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.

Index