Solving Problems with Python: COSC 101 — Intro. to Computing I

Release 1.0a

J. Sommers, Editor

January 04, 2013

Contents

Part I

COSC 101 - Introduction to Computing I

This book was originally written by A. Downey under the titles "How to think like a computer scientist", and "Python for software design." Under terms of the GNU Public License, his text has been remixed and modified for use in COSC 101, Intro to Computing I, at Colgate University. For the original source text, see <http://thinkpython.com>. See the Postscript for original book credits.

The source text for this book is written in reStructuredText and converted to pdf, ebook, and HTML using the excellent Sphinx documentation tool (<http://sphinx.pocoo.org>). All code and book source is available under terms of the GNU Public License and hosted at github.com <https://github.com/ColgateUniversityComputerScience>.

Contributors:

- 13. Hay
- A.D. Jaggard
- 10. Sommers

Current Editor:

- 10. Sommers

Version 2013.01.

Contents:

CHAPTER 1

The way of the program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating 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, 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 learn 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.

There are also **low-level languages**, sometimes referred to as "machine languages" or "assembly languages." Loosely speaking, computers can only execute programs written in low-level languages. So programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.

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

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

Two kinds of programs process high-level languages into low-level languages: **interpreters** and **compilers**. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.

A compiler reads the program and translates it completely before the program starts running. In this context, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

Figure 1.1: Running an "interpreted" program.

Once a program is compiled, you can execute it repeatedly without further translation.

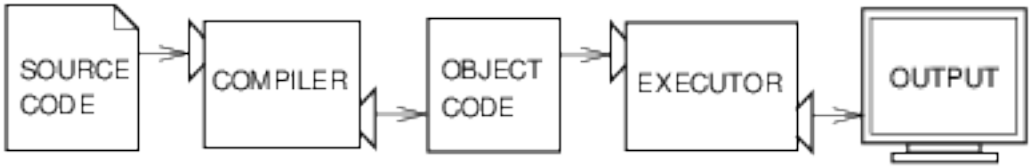


Figure 1.2: Running a "compiled" program.

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

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

The chevron, `>>>`, is the **prompt** the interpreter uses to indicate that it is ready. If you type `1 + 1`, the interpreter replies `2`.

Alternatively, you can store code in a file and use the interpreter to execute the contents of the file, which is called a **script**. By convention, Python scripts have names that end with `.py`.

In this course (COSC 101), we'll typically be writing programs within a program called IDLE. A screenshot of IDLE is shown below. (Note that this screenshot was taken on a Linux system, and that one Python statement was executed in the interpreter after starting it up. On your own computer, the window may look slightly different.)

This window is what you'll see once IDLE first starts up. This window is also referred to as the "shell". All that means is that it is this window in which you can directly interact with the Python interpreter. We'll frequently use this window to experiment with different Python statements and expressions and for very short (1–2 line) programs. For anything more than a few lines, you should write and save your program as a "script". To create a script using IDLE, simply go to the File menu and select "New window". In the window that opens, you can type your program. To execute (run) the program, you can go to the "Run" menu and select "Run module". The screenshot below shows IDLE with a 1-line script. Again, IDLE running on other computer systems may look slightly different.

There are other ways to execute Python scripts. For example, you can open a terminal window and type `python myprogram.py` (assuming you saved a script in the file `myprogram.py`). The details of opening a terminal window and executing Python scripts in this way vary among different operating systems (e.g., Linux, MacOS X, Windows, etc.). Normally, we will just be using IDLE in this course.

You are strongly encouraged to install Python on your own personal computer — it's free! There are two options for installing Python: the official package at python.org, and a package with a few additional (and useful) libraries at enthought.com. These are available at http://www.python.org/download/ and http://enthought.com/repo/free/. It is recommended that you install the Enthought version, since different class exercises may make use of the additional libraries available with this distribution.

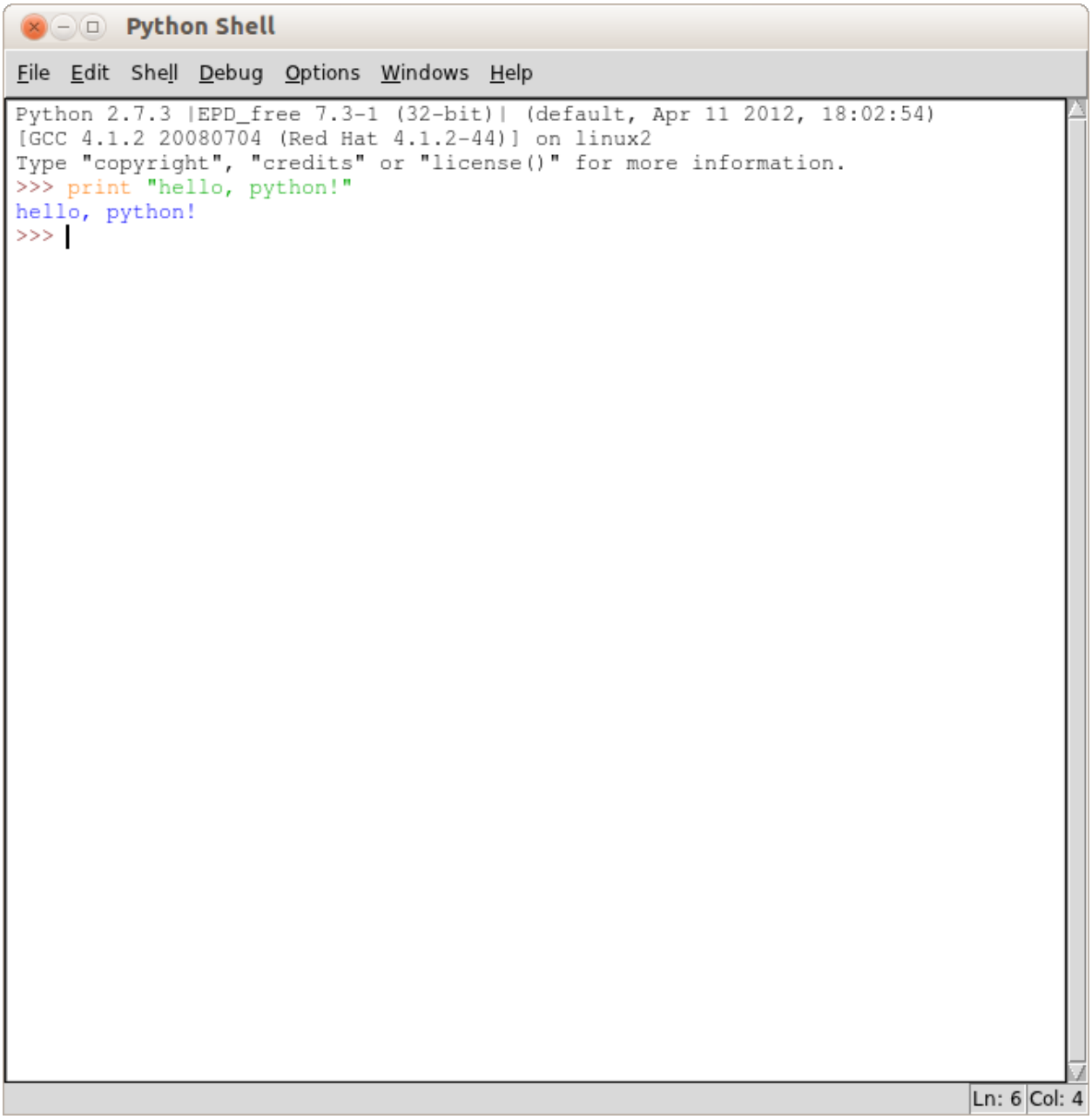


Figure 1.3: Main IDLE window (the “shell”).

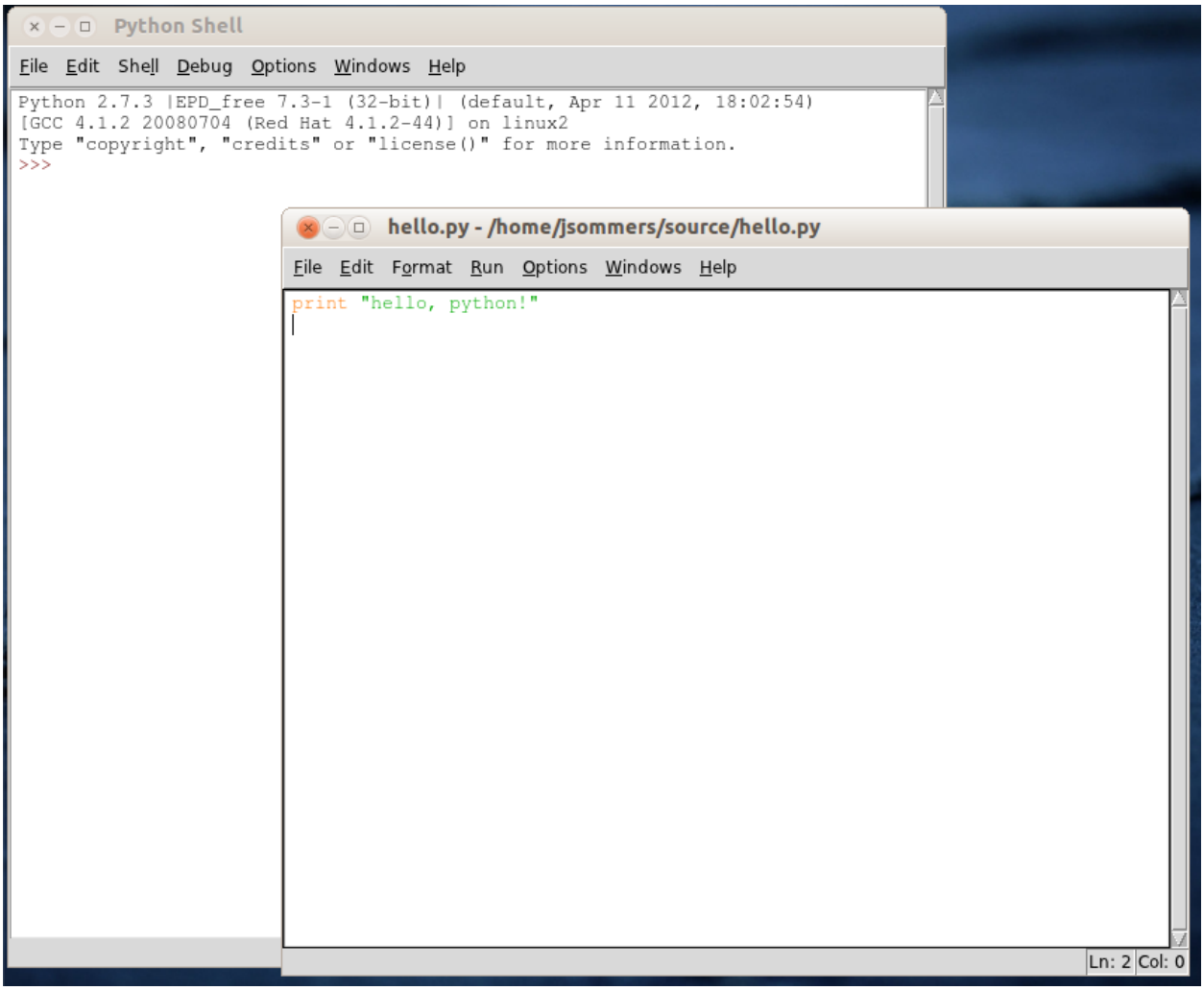


Figure 1.4: IDLE with a script window.

1.2 What is a program?

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

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

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

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

math: Perform basic mathematical operations like addition and multiplication.

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

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

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as *the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions*.

That may be a little vague, but we will come back to this topic when we talk about **algorithms**.

1.3 What is debugging?

Programming is error-prone. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**. Interestingly, one of the original uses of the term "bug" actually had to do with insects: an error in an early computer system was traced to a problem caused by a moth trapped in an electronic circuit! An early computer pioneer, Grace Hopper, is credited with having coined the term (read more about Grace Hopper on Wikipedia).

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

1.3.1 Syntax errors

Python can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a **syntax error**.

In English readers can tolerate most syntax errors, 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 display 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, you will make fewer errors and find them faster.

1.3.2 Runtime errors

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

1.3.3 Semantic errors

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

The problem is that the program you wrote is not the program you wanted to write. The meaning (semantics) of the program is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

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

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (A. Conan Doyle, *The Sign of Four*)

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

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux." (*The Linux Users's Guide* Beta Version 1).

Later chapters will make more suggestions about debugging, creating "tests" to ensure that your programs behave as expected, and other programming practices.

1.4 Formal and natural languages

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

Formal languages are languages that are designed by people for 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 + = 3 $ 6` is not. H_2O is a syntactically correct chemical formula, but ${}_2Zz$ is not.

Syntax rules come in two flavors, pertaining to *tokens* and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with `3 + = 3 $ 6` is that `$` is not a legal

token in mathematics (at least as far as I know). Similarly, $_2$Zz 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 + = 3 $ 6` is illegal because even though + and = are legal tokens, you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

Example:

1. Write a well-structured English sentence with invalid tokens in it. Then write another sentence with all valid tokens but with invalid structure.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The penny dropped," you understand that "the penny" is the subject and "dropped" is the predicate. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a penny is and what it means to drop, 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 some 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 penny dropped," there is probably no penny and nothing dropping. [1] 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, the details matter. Small errors in spelling and 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 you write in a new language is called "Hello, World!" because all it does is display the words, "Hello, World!" In Python, it looks like this:

[1] This idiom means that someone realized something after a period of confusion.

```
print 'Hello, World!'
```

This is an example of a **print statement** [2], which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result is the words

```
Hello, World!
```

The quotation marks in the program mark the beginning and end of the text to be displayed; they don't appear in the result.

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

1.6 Debugging

It is a good idea to read this book in front of a computer so you can try out the examples as you go. You can run most of the examples in interactive mode, but if you put the code into a script, it is easier to try out variations.

Example:

1. Whenever you are experimenting with a new feature, you should try to make mistakes. In the "Hello, world!" program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `print` wrong?

This kind of experiment helps you remember what you read; it also helps with debugging, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent or embarrassed.

There is evidence that people naturally respond to computers as if they were people. [3] When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people.

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a debugging section, like this one, with my thoughts about debugging. I hope they help!

1.7 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*.

[2] In Python 3.0, `print` is a function, not a statement, so the syntax is `print('Hello, World!')`. We will get to functions soon!

[3] See Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*.

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

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

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

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

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

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

prompt: Characters displayed by the interpreter to indicate that it is ready to take input from the user.

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

interactive mode: A way of using the Python interpreter by typing commands and expressions at the prompt.

script mode: A way of using the Python interpreter to read and execute statements in a script.

program: A set of instructions that specifies a computation.

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

bug: An error in a program.

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

syntax: The structure of a program.

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

exception: An error that is detected while the program is running.

semantics: The meaning of a program.

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

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

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

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

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

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

1.8 Exercises

1. Ways to get help in Python. These two quasi-exercises show you ways in which you can get help on different statements and operations in Python.

   (a) Use a web browser to go to the Python website http://python.org. This page contains information about Python and links to Python-related pages, and it gives you the ability to search the Python documentation.

   For example, if you enter `print` in the search window, the first link that appears is the documentation of the `print` statement. At this point, not all of it will make sense to you, but it is good to know where it is.

   (b) Start the Python interpreter and type `help()` to start the online help utility. Or you can type `help('print')` to get information about the `print` statement.

2. Start the Python interpreter and use it as a calculator. Python's syntax for math operations is almost the same as standard mathematical notation. For example, the symbols +, − and / denote addition, subtraction and division, as you would expect. The symbol for multiplication is $\star$.

   (a) If you run a 10 kilometer race in 43 minutes 30 seconds, what is your average time per mile? What is your average speed in miles per hour? (Hint: there are 1.61 kilometers in a mile).

   (b) How many seconds are there in 8 weeks? Write a one-line Python program to print the answer.

CHAPTER 2

Variables, expressions and statements

2.1 Values and types

A **value** is one of the basic things a program works with, like a letter or a number. The values we have seen so far are the numbers `1`, and `2`, and the phrase `'Hello, World!'`.

These values belong to different **types**: `2` is an integer, and `'Hello, World!'` is a **string**, so-called because it contains a "string", or sequence, of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks. (Note that you can use both single and double quotes to enclose a Python string. More on strings soon.)

The print statement works as you might expect for integers.

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

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called **floating-point**.

```
>>> type(3.2)
<type 'float'>
```

What about values like `'17'` and `'3.2'`? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

They're strings.

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 still legal:

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

Well, that's not what we expected at all! Python interprets `1,000,000` as a comma-separated sequence of integers, which it prints with spaces between.

This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

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. Another way to think of a variable is as a *named storage location in memory*.

An **assignment statement** creates new variables and gives them values:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

This example makes three assignments. The first assigns a string to a new variable named `message`; the second gives the integer `17` to `n`; the third assigns the (approximate) value of π to `pi`.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the previous example:

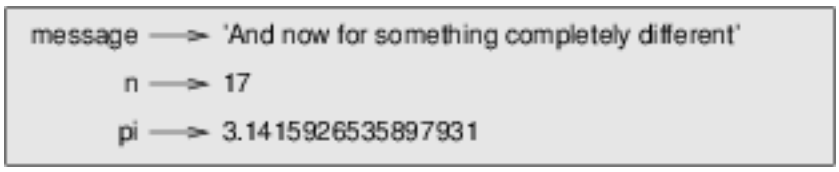


Figure 2.1: Variable assignment examples.

To display the value of a variable, you can use a print statement:

```
>>> print n
17
>>> print pi
3.14159265359
```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>

**Example**:

1. If you type an integer with a leading zero, you might get a
   confusing error:
```

```
                zipcode = 02492 ^ SyntaxError: invalid token

Other numbers seem to work, but the results are bizarre:

::

    >>> zipcode = 02132
    >>> print zipcode
    1114

Can you figure out what is going on? Hint: print the values ``01``,
``010``, ``0100`` and ``01000``.
```

2.3 Variable names and keywords

Programmers generally choose names for their variables that are meaningful — they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter. (Beginning a variable name with an upper-case letter is often only done is certain situations, which we won't see until a bit later.)

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

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

It turns out that `class` is one of Python's **keywords**. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python version 2 (which is what we're using) has 31 keywords [1]:

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

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.

[1] In Python 3.0, `exec` is no longer a keyword, but `nonlocal` is.

2.4 Statements

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.

A script usually contains an ordered sequence of statements. If there is more than one statement, the results appear one at a time as the statements *execute*, or run. In all the scripts we will see for now, statements execute one after another, top to bottom, just as you would normally read a page in English.

For example, the script

```
print 1
x = 2
print x
print 1,x
```

produces the output

```
1
2
1 2
```

The assignment statement produces no output. The last statement prints the literal integer `1` followed by a space, followed by the value referred to by the variable `x`. You can use commas to separate items to be printed in a `print` statement; Python will include spaces between each item (and the commas will not be shown in the output).

2.5 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called **operands**.

The operators `+`, `-`, `*`, `/` and `**` perform addition, subtraction, multiplication, division and exponentiation, as in the following examples:

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

In some other languages, `^` is used for exponentiation, but in Python it is a bitwise operator called XOR. Bitwise operators will not be covered in this book, but you can read about them at http://wiki.python.org/moin/BitwiseOperators.

The division operator might not do what you expect:

```
>>> minute = 59
>>> minute/60
0
```

The value of `minute` is 59, and in conventional arithmetic 59 divided by 60 is 0.98333, not 0 . The reason for the discrepancy is that Python is performing **floor division** [2].

When both of the operands are integers, the result is also an integer; floor division chops off the fraction part, so in this example it rounds down to zero.

[2] In Python 3.0, the result of this division is a `float`.

If either of the operands is a floating-point number, Python performs floating-point division, and the result is a `float`:

```
>>> minute/60.0
0.98333333333333328
```

If you *want* floor division and you are working with floating point operands, you can use the `//` operator to perform integer division:

```
>>> minute // 60.0
0.0
```

The resulting type from `//` follows the same rule as with the "regular" division operator (`/`): if both operands are integers, the result is an integer, and if at least one operand is floating point, the result is floating point.

2.6 Expressions

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

```
17
x
x + 17
```

If you type an expression in interactive mode, the interpreter **evaluates** it and displays the result:

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

But in a script, an expression all by itself doesn't do anything! If you want to show output to the screen, you must use a `print` statement. This is a common source of confusion for beginners.

Examples:

1. Type the following statements in the Python interpreter to see what they do:

   ```
   5
   x = 5
   x + 1
   ```

2. Now put the same statements into a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

2.7 Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

- **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. 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 if it doesn't change the result.
- **E**xponentiation has the next highest precedence, so `2**1+1` is 3, not 4, and `3*1**3` is 3, 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` is 5, not 4, and `6+4/2` is 8, not 5.

- Operators with the same precedence are evaluated from left to right. So in the expression `degrees / 2 * pi`, the division happens first and the result is multiplied by `pi`. To divide by 2π, you can use parentheses or write `degrees / 2 / pi`.

2.8 String operations

As mentioned above, strings in Python can be enclosed in single or double quotes. The following two statements are equivalent:

```
mystring = 'green eggs and spam'
mystring = "green eggs and spam"
```

Sometimes you might need to create really long strings that span multiple lines. You can use yet another quoting method for that: *triple* quotes (three single quotes in a row). For example:

```
    mystring = '''
green
eggs
and
spam
'''
```

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

```
'2'-'1'    'eggs'/'easy'    'third'*'a charm'
```

The + operator works with strings, but it might not do what you expect: it performs **concatenation**, which means joining the strings by glueing them end-to-end. For example:

```
first = 'throat'
second = 'warbler'
print first + second
```

The output of this program is `throatwarbler`.

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

This use of + and * makes sense by analogy with addition and multiplication. Just as `4*3` is equivalent to `4+4+4`, we expect `'Spam'*3` to be the same as `'Spam'+'Spam'+'Spam'`, and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition has that string concatenation does not?

2.9 Reassignment

It is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

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

The output of this program is `5 7`, because the first time `bruce` is printed, its value is 5, and the second time, its value is 7. The comma at the end of the first `print` statement suppresses the newline, which is why both outputs appear on the same line.

Here is what **reassignment** looks like in a state diagram:

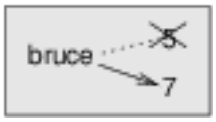


Figure 2.2: Reassignment state diagram.

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

First, equality is a symmetric relation 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 either true or false, 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 does not change the value of `b`, so they are no longer equal.

Although reassignment is frequently helpful, you should use it with care. If the values of variables change frequently, it can make the code difficult to read and debug.

You can also make multiple assignments on the same line:

```
a = b = c = 5
# a, b, and c each refer to the integer value 5
```

You can read such a statement from right to left: the value of 5 is assigned to `c`, which is also assigned to `b`, which is also assigned to `a`.

2.10 Updating variables

One of the most common forms of assignment is an **update**, where the new value of the variable depends on the old.

```
x = x+1
```

This means "get the current value of `x`, add one, and then update `x` with the new value."

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to `x`:

```
>>> x = x+1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>> x = 0
>>> x = x+1
```

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

There are "convenience operators" built in to Python that make updating variables slightly simpler, syntactically. For example:

```
x += 2
```

does the same thing as

```
x = x + 2
```

with a couple fewer keystrokes. Similar to +=, there are −=, ⋆=, and /= convenience operators. Many Pythonistas use these convenience operators, so if you look at code posted on, for example, *Stack Overflow* (http://stackoverflow.com/questions/tagged/python), you'll often see these operators.

2.11 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is 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 start 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 # percentage of an hour
```

Everything from the # to the end of the line is ignored—it has no effect on the program.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.

This comment is redundant with the code and useless:

```
v = 5       # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5       # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff. It will be important in all the programs you create in this class (even short programs!) to carefully consider *variable name choices* and to *include useful comments*. Both of these items are considered part of the **documentation** of a program, and help humans (you and others) to better understand the purpose and intent of a program.

2.12 Debugging

At this point the syntax error you are most likely to make is an illegal variable name, like `class` and `yield`, which are keywords, or `odd~job` and `US$`, which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

For syntax errors, the error messages don't help much. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

The runtime error you are most likely to make is a "use before def"; that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Variables names are case sensitive, so `LaTeX` is not the same as `latex`.

At this point the most likely cause of a semantic error is the order of operations. For example, to evaluate $\frac{1}{2\pi}$, you might be tempted to write

```
>>> 1.0 / 2.0 * pi
```

But the division happens first, so you would get $\pi/2$, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

Exercise:

1. Rewrite the last expression (`1.0 / 2.0 * pi`) so that the correct answer is reached.

2.13 Glossary

value: One of the basic units of data, like a number or string, that a program manipulates.

type: A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

integer: A type that represents whole numbers.

floating-point: A type that represents numbers with fractional parts.

string: A type that represents sequences of characters.

variable: A name that refers to a value.

statement: A section 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.

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

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

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.

floor division: The operation that divides two numbers and chops off the fraction part.

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

evaluate: To simplify an expression by performing the operations in order to yield a single value.

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

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

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

update: An assignment where the new value of the variable depends on the old.

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

2.14 Exercises

1. Assume that we execute the following assignment statements:

   ```
   width = 17
   height = 12.0
   delimiter = '.'
   ```

   For each of the following expressions, write the value of the expression and the type (of the value of the expression).

   (a) `width/2`

   (b) `width/2.0`

   (c) `height/3`

   (d) `1 + 2 * 5`

   (e) `delimiter * 5`

   Use the Python interpreter to check your answers.

2. Practice using the Python interpreter as a calculator:

   (a) The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with radius 5? Hint: 392.6 is wrong!

   (b) Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?

   (c) If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?

3. Consider the following two variable assignments:

   ```
   panicstring = "don't panic"
   answer = 42
   ```

   Write a Python statement to print the following: `don't panic -- think 42`

4. Consider a modern computer processor that runs at 2 GHz, which simply means "2 billion cycles per second". If the speed of light is 299, 792, 458 meters per second, how many centimeters does light travel in the time it takes a processor to execute one cycle? Write a one-line Python program to print the answer.

CHAPTER 3

Using functions

In this chapter we'll learn about how to use functions that are built in to Python. Python contains hundreds, if not thousands, of *modules* that contain many functions that you can use in your programs. Over the span of this course we will use quite a few different modules and functions. This chapter is about how to tap into these features.

3.1 Function calls

In the context of programming, a **function** is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a **function call**:

```
>>> type(32)
<type 'int'>
```

The name of the function is `type`. The expression in parentheses is called the **argument** of the function. The result, for this function, is the type of the argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the **return value**.

3.2 Type conversion functions

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

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

`int` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` converts integers and strings to floating-point numbers:

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

Finally, `str` converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.3 Getting user input: the `input` and `raw_input` functions

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 a built-in function called `raw_input` that gets input from the keyboard. When this function is called, the program stops and waits for the user to type something. When the user presses `Return` or `Enter`, the program resumes and `raw_input` returns what the user typed as a string.

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

Before getting input from the user, it is a good idea to print a prompt telling the user what to input. `raw_input` can take a prompt as an argument:

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

The sequence `\n` at the end of the prompt represents a **newline**, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to `int`:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
```

But if the user types something other than a string of digits, you get an error:

```
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int()
```

We will see how to handle this kind of error later.

3.4 More on strings

A string is a **sequence** of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'apple'
>>> letter = fruit[1]
```

The second statement selects character number 1 from `fruit` and assigns it to `letter`.

The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
>>> print letter
p
```

For most people, the first letter of `'apple'` is `a`, not `p`. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

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

So `a` is the 0th letter ("zero-eth") of `'apple'`, `p` is the 1th letter ("one-eth"), and `p` is the 2th ("two-eth") letter.

Since indices start at 0, we can see that the last valid index of the string `'apple'` is 4, which is one less than the length of the string (which is 5 characters).

	a	p	p	l	e
index	0	1	2	3	4

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

```
>>> fruit = 'apple'
>>> length = len(fruit)
>>> print length
5
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in `'apple'` with the index 5. To get the last character, you have to subtract 1 from the length of the string.

A built-in Python function that we'll use frequently with sequence types like strings is `len`. This function returns the number of items in the sequence as an integer. It can conveniently be used to access the last character of a string, no matter the length of the string:

```
>>> fruit = 'coconut'
>>> fruitlen = len(fruit)
>>> print fruitlen
7
>>> print fruit[fruitlen-1]
't'
```

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

Also, strings are *immutable*, which means that you cannot modify them once they're created. For example, if you try to modify one character of a string using an assignment statement, you'll get an error:

```
>>> fruit[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

In addition to using indices from 0 to one less than the length of a string, you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on. The following table shows both *positive* and *negative* indices, and how they can be used to access characters of a string.

	a	p	p	l	e
positive indices	0	1	2	3	4
negative indices	-5	-4	-3	-2	-1

3.5 Case study 1: printing out the characters of a string

As we've learned, *a string is a sequence of characters*. Sometimes, solving a problem requires us to inspect each character of a string, one by one. In other words, to inspect each character *in sequence*. Python includes the `for` statement to help with repetitive tasks like this. Here is an example that simply prints out each character by itself, in sequence:

```
fruit = "kiwi"
for char in fruit:
    print char
print "done!"
```

There are quite a few new syntactical elements to this example, so let's go through it in detail.

First, we see the `for` statement, which can be read in English as "for every item in the sequence", or "for every character in the string fruit". The last item of a `for` statement must, therefore, be a sequence type, like a string.

The `for` statement is usually called a "`for` loop", because of its repetitive nature. The effect of the program is to assign each letter of the string `fruit` to the variable `char`, one by one. For each assignment, the indented statement block underneath the `for` statement consisting of the line `print char` is executed. That is, the statement block is *repeated* for each character in the string. As a result, each letter of the string is printed by itself on separate lines. The complete output of the program is shown below:

So, the word `char` in the `for` statement is a variable that is assigned each item of the sequence, one-by-one. (Note that the variable name `char` is *just a variable name*, and has no inherent meaning to the `for` loop: we could have just as easily used `seed` or some other valid variable name.)

Also, notice that a colon (`:`) appears at the end of the `for` statement, and that the next line is indented. Whenever you encounter a colon at the end of a statement in Python, the next statement **must** be indented in order for the code to be syntactically correct. There can be more than one indented statement, but *at least* one must be present; these indented statements are referred to as a **statement block**.

```
k
i
w
i
done!
```

The sequence of operations that are executed by the 4-line program is not especially obvious just from glancing at the program. In order to be able to understand the output, we need to think about the meaning of the `for` statement, and

trace each action that the Python interpreter would make. Learning to read a program is an important skill to develop, and one that you will need to work on throughout this course.

In detail, the way that this program is sequentially executed by the Python interpreter is as follows:

1. The variable fruit is assigned the string `'kiwi'`
2. We encounter the `for` loop. The variable `char` is assigned the *first* letter of the string referred to by `fruit` (`'k'`)
    - We print `char`, which is just `'k'`
3. We go back to the top of the `for` loop; the next letter, `'i'`, is assigned to `char`
    - We print `char`, which is `'i'`
4. We go back to the top of the `for` loop; the next letter, `'w'`, is assigned to `char`
    - We print `char`, which is `'w'`
5. We go back to the top of the `for` loop; the last letter, `'i'`, is assigned to `char`
    - We print `char`, which is `'i'`
6. We exit the for loop since we have traversed every element in the sequence (string) `'kiwi'`. We print `'done!'` since that is the next statement in the program.

3.6 Math functions

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.

Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a **module object** named math. If you print the module object, you get some information about it:

```
>>> print math
<module 'math' (built-in)>
>>>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called `log` that computes logarithms base `e`.

The second example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression `math.pi` gets the variable `pi` from the math module. The value of this variable is an approximation of π, accurate to about 15 digits.

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

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

A few last notes about importing modules:

1. By convention any `import` statements should always go at the *top* of your programs.
2. There are a huge number of modules built-in to Python (take a look at http://docs.python.org/library/ if you wish). We'll really just scratch the surface on these built-in capabilities. As we proceed through the course, you'll learn how to decipher the online Python documentation to be able to take advantage of more of these modules.
3. There are other ways to import modules, which you'll see later in the course (and which you'll likely encounter if you look at Python code examples on the web).

3.7 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, the argument of a function can be any kind of expression, including arithmetic operators:

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

And even function calls:

```
x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error.

```
>>> minutes = hours * 60                    # right
>>> hours * 60 = minutes                    # wrong!
SyntaxError: can't assign to operator
```

3.8 Case study 2: string traversal using `range`, `len`, and a for loop

In the next two case studies, we'll get practice with function composition and learn two new built-in Python functions: `range` and `round`.

We already know that we can use a for loop to access each character of a string, one by one. We also know that we can use an *index* to access individual characters in a string by position. What if we wanted to combine these two ideas, and cycle through the valid indices of a string? For example, if we wanted to go through the integer values 0..4 to access

each character of the string `apple` by index. Well, Python has a built-in function named `range` that can help with exactly that task. For example:

```
>>> range(4)
[0, 1, 2, 3]
>>>
```

The `range` function takes an integer as a parameter, and returns a sequence of integers from 0 through the supplied argument minus 1. (The sequence of integers that `range` returns is called a `list` in Python. A `list` is a sequence type with some similarities to strings. We'll learn more about lists soon.)

We can use the `range` function in a `for` loop to print the integers from 0 through 3 as follows:

```
for index in range(4):
    print index
```

```
0
1
2
3
```

Now, to solve the problem of printing the characters of a string *by index*, we can *compose* the `range` and `len` functions in a `for` loop, as follows:

```
fruit = 'kiwi'
for index in range(len(fruit)):
    print index, fruit[index]
```

In the statement block inside the `for` loop, we print both the value of the variable `index`, and the character at the given index in the string. Thus, the output should be:

```
0 k
1 i
2 w
3 i
```

The composition of `range` and `len` in a `for` loop is quite powerful! It's also a good Pythonic idiom to understand: we'll use it often in `for` loop construction.

One last note about the `range` function: it can actually take more than one argument to flexibly construct a variety of different numeric sequences. We'll learn about this more complex use of `range` a bit later.

3.9 Case study 3: making a table of square roots

In the last case study of this chapter, we'll again use `range` in a `for` loop to print tables of numeric values. Say, for example, that we want to make a table of square roots, like:

```
The square root of 0 is 0
The square root of 1 is 1
The square root of 2 is ?!
...
```

I don't remember a good approximation to the square root of two off the top of my head, but I bet we can coerce Python into telling us! Here's one way how:

```
import math

print "My amazing table of square roots!"
```

```
for number in range(5):
    print "The square root of", number, "is", math.sqrt(number)
```

The output of our program should be:

```
My amazing table of square roots!
The square root of 0 is 0.0
The square root of 1 is 1.0
The square root of 2 is 1.41421356237
The square root of 3 is 1.73205080757
The square root of 4 is 2.0
```

Make sure you understand how Python produces the above output.

Hmm... we've got a table of square roots, but some of the values are a little unwieldy because of so many decimal places. To make the output look a little nicer, we can use the built-in Python `round` function. `round` takes two arguments: a value to round, and the number of decimal places to which to round the number. If we want to round to 2 decimal places, we can modify the above program to compose the `round` and `math.sqrt` functions:

```
import math

print "My super-amazing table of square roots!"
for number in range(5):
    print "The square root of", number, "is", round(math.sqrt(number),2)
```

```
My super-amazing table of square roots!
The square root of 0 is 0.0
The square root of 1 is 1.0
The square root of 2 is 1.41
The square root of 3 is 1.73
The square root of 4 is 2.0
```

Ah. That's better. Later, we'll learn ways to make our output look even nicer, but for now, `round` does a pretty good job.

3.10 Debugging

There are a few common error patterns and issues to be aware of related to functions and new syntax we've seen in this chapter.

1. Using a function in an external module, but forgetting to use `import`:

   ```
   >>> print math.sqrt(10)
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   NameError: name 'math' is not defined
   ```

   To fix this problem, just be sure to say `import math` at the top of your program.

2. Constructing a for loop that results in the statement block not being executed.

   ```
   mystring = ''
   for char in mystring:
       print "The new phone books are here!"
   ```

   The result of this program is ... nothing! The reason is that the string, while valid, is "empty". Thus, there are no characters to be assigned to the variable `char`, and we won't execute the statement block within the `for` loop. Nothing will happen.

Some other questions for you to consider along the same lines as this error pattern: what happens if you call `len` with an empty string? What happens if you use `range` with a negative number?

Thankfully, it's *impossible* to create a `for` loop that never stops. We will, however, encounter another *iteration* mechanism in Python where "infinite loops" are possible.

3. Using a non-sequence type for the last part of the `for` statement.

   The last part of the `for` statement has to be a sequence type, or something that is *iterable*. If not, you'll get an error like the following:

```
>>> for value in 5:
...     print value
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>>
```

   All this error says is that the integer value 5 isn't a sequence type, so the `for` loop blew up.

More generally, when Python crashes, the "Traceback" message that it shows contains a lot of information. The most useful parts are usually:

- What kind of error it was, and
- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
SyntaxError: invalid syntax
```

In this example, the problem is that the second line is indented by one space. But the error message points to `y`, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is

$$SNR_{db} = 10 \times \log_{10}(P_{signal}/P_{noise})$$

In Python, you might write something like this:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power / noise_power
decibels = 10 * math.log10(ratio)
print decibels
```

But when you run it, you get an error message:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
```

```
    decibels = 10 * math.log10(ratio)
OverflowError: math range error
```

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of `ratio`, which turns out to be 0. The problem is in line 4, because dividing two integers does floor division. The solution is to represent signal power and noise power with floating-point values.

So in general, error messages tell you where the problem was discovered, but that is often not where it was caused.

3.11 Glossary

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

function call: A statement that executes a function. It consists of the function name followed by an argument list.

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

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

import statement: A statement that reads a module file and creates a module object.

module object: A value created by an `import` statement that provides access to the values defined in a module.

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

sequence: An ordered set; that is, a set of values where each value is identified by an integer index.

item: One of the values in a sequence.

index: An integer value used to select an item in a sequence, such as a character in a string.

loop: A part of a program that can execute repeatedly.

composition: Using an expression as part of a larger expression, or a statement as part of a larger statement.

3.12 Exercises

1. Ask for a three-character string from a user, then construct and print a new string by swapping the first and last characters of the string entered by the user. (You can assume that the user always types a string consisting of 3 letters.) For example, if the user types `'box'`, your program should print `'xob'`. Note: using the `if` statement (discussed in the next chapter) is off limits!
2. Ask for a string from the user. Print the string right-justified within a page width of 40 characters. For example, if a user types `'abecedarian'`, which is 11 characters long, your program should print exactly 29 spaces followed by `'abecedarian'` (i.e., the total width of what you print should be exactly 40 characters). You can assume that the string entered by the user is at most 40 characters long.
3. Construct a short program with a `for` loop to print the values of the sequence 25, 50, 75, ... 175, 200. Your `for` loop should use `range` with just one argument.
4. Write a program that asks a user for a positive integer, then prints a table of cubes from 1 through that number. Make the table output as nice as you can using what we've covered so far. For example, if a user enters the number 3, your program should print the numbers 1, 8, and 27 (1^3, 2^3, and 3^3) in a nice table.

5. Write a program that asks a user for a string, then prints the characters of the string *in reverse*, one on each line. For example, if a user enters the string `'magic'`, your program should print:

```
c
i
g
a
m
```

6. Modify the program to make a table of square roots by asking the user for the largest number for which to compute the square root. For example, if the user types `11`, your table should show the square roots from 0 through 11, including both end points.

7. Write a program to compute the square root of the sum of numbers from 1 to 1000. You should use a `for` loop to compute the sum, and the `sqrt` function in the `math` module to compute the square root.

8. The interest earned on an investment can be computed as `interest = principal * rate`.

   Write a program that asks a user for an interest rate as a floating point number, an investment amount as a floating point number, and the number of years. Your program should print, for each year, the current amount of the principal. Note to economists and mathematicians: you should *not* use the exponential formula for this problem.

```
What is the interest rate? 0.05
What is the principal (amount invested)? 100
How many years? 5
After 1 years, the principal is 105.0
After 2 years, the principal is 110.25
After 3 years, the principal is 115.76
After 4 years, the principal is 121.55
After 5 years, the principal is 127.63
```

CHAPTER 4

Making decisions

4.1 Boolean expressions

Solving almost any problem requires that we make *decisions*, and at the heart of decision-making within programs are **Boolean** values and expressions. A **boolean expression** is one that is either true or false. The following examples use the operator ==, which compares two operands and produces `True` if they are equal and `False` otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` and `False` are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

The == operator is one of the **relational operators**; the others are:

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

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

You can use the relational operators to check whether a value is within a numerical range:

```
>>> x = 4
>>> 3 <= x < 5
True
```

Besides applying these relational operators to numerical operands, they can also be used with strings. To see if two strings are equal:

```
if word == 'banana':
    print  'All right, bananas.'
```

Other relational 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:
    # must be equal!
    print 'All right, bananas.'
```

Python does not handle uppercase and lowercase letters the same way that people do: all the uppercase letters come before all the lowercase letters. When we look at strings in more detail in a later chapter, we'll revisit this issue.

Lastly, there is a Boolean operator called `in` that can be used with sequence types in Python to test whether one sequence is contained within another. The most common use for in with strings is to test whether a character or substring is contained within a string. For example, both of the following Boolean expressions would evaluate to `True`:

```
'a' in 'banana'
'nana' in 'banana'
```

But this one would evaluate to `False`:

```
'A' in 'banana'
```

4.1.1 Modulus operator

The **modulus operator** is not specifically related to Boolean variables or making decisions, but will be useful in various problems that we'll soon encounter. The modulus (or "mod") operator works on integers 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
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

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

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

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

4.1.2 Logical operators

There are three **logical operators**: `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 0 *and* less than 10.

`n%2 == 0 or n%3 == 0` is `True` if *either* of the conditions is `True`, that is, if the number is divisible by 2 *or* 3.

Finally, the `not` operator negates a boolean expression, so `not (x > y)` is `True` if `x > y` is `False`, that is, if `x` is less than or equal to `y`.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as `True`.

```
>>> 17 and True
True
```

This flexibility can be useful, but there are some subtleties to it that can be confusing. You should avoid exploiting this behavior as it tends to make programs more difficult to read and understand.

4.1.3 Conditional execution

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

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

The boolean expression after the `if` statement is called the **condition**. If it is `True`, then the indented statement gets executed. If not, nothing happens.

`if` statements have the same structure as the `for` statement: a header with a colon at the end, followed by an indented body. Statements like this are called **compound statements**. When ever you have a colon at the end of a program statement, IDLE will automatically place your cursor at a properly indented location on the next line. If you need to do "manual" indentation, the convention in Python is to use 4 spaces (and no tabs).

Again, for compound statements, there is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

```
if x < 0:
    pass          # need to handle negative values!
```

4.1.4 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 this:

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

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays a message to that effect. If the condition is `False`, the second set of statements is executed. Since the condition must be `True` or `False`, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution.

4.1.5 Case study: simulating a coin toss

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be **deterministic**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are 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 use algorithms that generate **pseudorandom** numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom numbers (which we'll simply call "random" from now on).

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

```
import random

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

The function `randint` takes parameters `low` and `high` and returns an integer between `low` and `high` (including both).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

So, one way to simulate tossing a fair coin (i.e., with equal chance it comes up heads or tails), we could use the following program:

```
import random

if random.randint(0,1) == 0:
    print "Heads!"
else:
    print "Tails!"
```

The `randint` function will return 0 or 1 with equal probability, so that will effectively simulate a coin toss. Alternatively, we could use the `random` function to do the same thing:

```
import random

if random.random() < 0.5:
    print "Heads!"
else:
    print "Tails!"
```

4.1.6 Chained conditionals

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

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

`elif` is an abbreviation of "else if". Again, *exactly one branch will be executed.* There is no limit on the number of `elif` statements. If there is an `else` clause, it has to be at the end, but there doesn't have to be one.

```
import random

rps = random.randint(1,3)
if rps == 1:
    print "Rock!"
elif rps == 2:
    print "Paper!"
elif rps == 3:
    print "Scissors!"
```

Each condition is checked in order. If the first is `False`, the next is checked, and so on. If one of them is `True`, the corresponding branch executes, and the statement ends. Even if more than one condition is `True`, only the first `True` branch executes.

4.1.7 Nested conditionals

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

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

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print 'x is a positive single-digit number.'
```

The `print` statement is executed only if we make it past both conditionals, so we can get the same effect with the `and` operator:

```
if 0 < x and x < 10:
    print 'x is a positive single-digit number.'
```

4.1.8 Debugging

> *The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.*
>
> Brian Kernighan, "Unix for Beginners" (1979)

Debugging problems can become a bit more complex when dealing with conditional statements. To understand what is causing a problem in a program, we often want to know which *branch* is being taken. To do that, it can be helpful

to insert `print` statements within `if`, `elif`, and `else` statement blocks. Adding `print` statements to help reveal what the program is doing is often referred to as **trace debugging**, **printf debugging**, or **caveman debugging**. (The term "printf debugging" comes from the `printf` function in the C language, which is fairly similar to the `print` statement in Python.) Although the term "caveman" doesn't cast a particularly favorable light on this technique, it is nonetheless an extremely useful and widely used method for understanding what a program is doing.

4.1.9 Glossary

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

boolean expression: An expression whose value is either `True` or `False`.

relational operator: One of the operators that compares its operands: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

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

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

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

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

branch: One of the alternative sequences of statements in a conditional statement.

chained conditional: A conditional statement with a series of alternative branches.

nested conditional: A conditional statement that appears in one of the branches of another conditional statement.

deterministic: Pertaining to a program that does the same thing each time it runs, given the same inputs.

pseudorandom: Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.

4.1.10 Exercises

1. Consider the following program:

```
i = input("Gimme a number: ")
if i == 0:
    print "You entered zero"
if i == 1:
    print "You entered one"
else:
    print "You entered something other than zero or one"
```

   Given an input of `0`, what will the program print? Why?

2. Write a program that asks for the number of a year (e.g., 1982) and prints whether that year was a leap year or not. A year is a leap year if it is evenly divisible by 4. If a year is also evenly divisible by 100 it is *not* a leap year, unless it is evenly divisible by 400 as well.

   For example, 1980 and 2012 were leap years. 1900 was *not* a leap year (evenly divisible by 100), but 2000 was (evenly divisible by 100 *and* 400).

3. Write a short program to play one round of rock-paper-scissors. Ask the user to enter 0, 1, or 2 to correspond to rock, paper, and scissors. Use the `random` module to have the computer randomly choose rock, paper, or scissors. Print a message indicating who wins, or whether there was a tie.

For those who haven't played rock, paper, scissors before (and even if you have), read the Wikipedia page for detail on related games, and programs (and robots) that have been built to play RPS: http://en.wikipedia.org/wiki/Rock-paper-scissors.

CHAPTER 5

Functions in depth

5.1 Adding new functions

So far, we have only been *using* the functions that come with Python, but it is also possible to add new functions. A **function definition** specifies the name of a new function and the sequence of statements to execute when the function is called.

Here is an example:

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print "I sleep all night and I work all day."
```

`def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments.

The first line of the function definition is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces. The body can contain any number of statements. We've now seen three *compound statements* that can include indented statement blocks: the `for` statement, the `if` statement, and now the `def` statement.

If you type a function definition in interactive mode, the interpreter prints ellipses (...) to let you know that the definition isn't complete:

```
>>> def print_lyrics():
...     print "I'm a lumberjack, and I'm okay."
...     print "I sleep all night and I work all day."
...
```

To end the function, you have to enter an empty line (this is not necessary in a script).

Defining a function creates a variable with the same name.

```
>>> print print_lyrics
<function print_lyrics at 0xb7e99e9c>
>>> print type(print_lyrics)
<type 'function'>
```

The value of `print_lyrics` is a **function object**, which has type `'function'`. The output from the `print print_lyrics` statement shows a number in hexadecimal format (the number starting `"0x..."`). The number refers to the *memory location* of the function object (i.e., where that function exists in Python's memory space).

The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that's not really how the song goes.

5.2 Definitions and uses

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

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print "I sleep all night and I work all day."

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. 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. In other words, the function definition has to be executed before the first time it is called.

Examples:

1. Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.
2. Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program? Why?

5.3 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up exactly where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good 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, you don't always want to read from top to bottom. Usually, you want to follow the flow of execution and read the program as Python would interpret it.

5.4 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called **parameters**. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
    print bruce
    print bruce
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

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

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
```

```
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions `'Spam '*4` and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

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

5.5 Variables and parameters are local

When you create a variable inside a function, it is **local**, which means that it only exists inside the function. For example:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

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

```
>>> print cat
NameError: name 'cat' is not defined
```

Parameters are also local. For example, outside `print_twice`, there is no such thing as `bruce`.

5.6 Return values

5.6.1 The special type `None`

Some of the built-in functions we have used, such as the math functions, produce results. When you call a function that returns a result, like `math.sqrt`, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.2360679774997898
```

But in a script, if you call a function that returns a result all by itself, the return value is lost forever, and does not even show up in the console as output!

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store or display the result (i.e., there is no `print` statement), it is not very useful.

Functions that do not return anything (also called "void functions") might display something on the screen or have some other effect, but they don't explicitly pass back a result. However, Python will implicitly return the special value `None`. Say `print_twice` is defined as follows:

```
def print_twice(s):
    print s
    print s
    # nothing returned from this function
```

In the interactive interpreter, we call the function and assign its result to the variable `result`:

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result
None
```

The value `None` is not the same as the string `'None'`. It is a special value that has its own type:

```
>>> print type(None)
<type 'NoneType'>
```

5.6.2 Functions with return values

If we want a function to hand back a result to the caller of the function, we can use the `return` statement with an expression. For example, the following function `area` returns the area of a circle with a given radius:

```
import math

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

The `return` statement means: *"Return immediately from this function and use the following expression as a return value."* The expression 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 absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Since these `return` statements are in an alternative conditional, only one will be executed.

As soon as a return statement executes, the function terminates without executing any subsequent statements. Code that appears after a `return` statement, or any other place the flow of execution can never reach, is called **dead code**.

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

```
# warning: this is problematic code!
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

This function is incorrect because if `x` happens to be 0, neither condition is true, and the function ends without hitting a `return` statement. If the flow of execution gets to the end of a function, the return value is `None`, which is not the absolute value of 0.

```
>>> print absolute_value(0)
None
```

By the way, Python provides a built-in function called `abs` that computes absolute values.

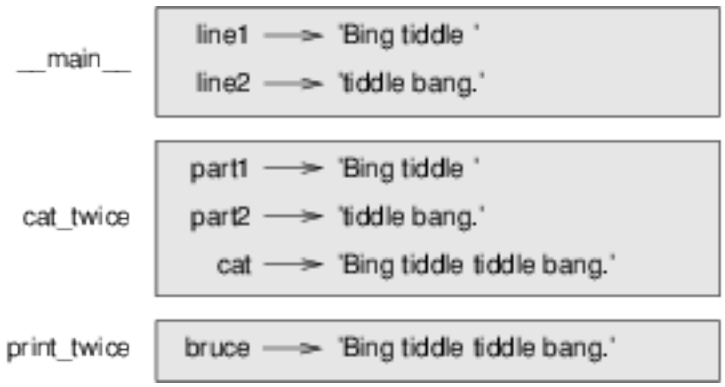
Example:

1. Write a `compare` function that returns `1` if `x > y`, `0` if `x == y`, and `-1` if `x < y`.

5.7 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 **frame** (or "stack frame"). A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:



The frames are arranged in a stack that indicates which function called which, and so on. In this example, `print_twice` was called by `cat_twice`, and `cat_twice` was called by `__main__`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `__main__`.

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `bruce` has the same value as `cat`.

If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called *that*, all the way back to `__main__`.

For example, if you try to access `cat` from within `print_twice`, you get a `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print cat
NameError: name 'cat' is not defined
```

This list of functions is called a **traceback**. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error (i.e., line 9, in the function `print_twice`).

The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom. [1]

5.8 Boolean functions

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

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

It is common to give boolean functions names that sound like yes/no questions; `is_divisible` returns either `True` or `False` to indicate whether `x` is divisible by `y`.

Here is an example:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

The result of the == operator is a boolean, so we can write the function more concisely by returning it directly:

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

Boolean functions are often used in conditional statements:

```
if is_divisible(x, y):
    print 'x is divisible by y'
```

It might be tempting to write something like:

```
if is_divisible(x, y) == True:
    print 'x is divisible by y'
```

But the extra comparison is unnecessary.

Example:

[1] Stack diagrams can either be drawn starting from the top, working down, or from the bottom, working up. It depends which hemisphere (northern or southern) you come from. (Just kidding. As long as you're consistent, it doesn't matter which way you draw it.)

1. Write a function `is_between(x, y, z)` that returns `True` if $x \leq y \leq z$ or `False` otherwise.

5.9 Debugging

5.9.1 Adding `print` statements

Breaking a large program into smaller functions creates natural checkpoints for debugging. If a function is not working, there are three possibilities to consider:

- There is something wrong with the arguments the function is getting.
- There is something wrong with the function.
- There is something wrong with the return value or the way it is being used.

To rule out the first possibility, you can add a `print` statement at the beginning of the function and display the values of the parameters (and maybe their types). Or you can write code that checks the preconditions explicitly.

If the parameters look good, add a `print` statement before each `return` statement that displays the return value. If possible, check the result by hand. Consider calling the function with values that make it easy to check the result.

If the function seems to be working, look at the function call to make sure the return value is being used correctly (or used at all!).

5.9.2 Unit testing and `assert`

After you think you've solved a problem, *how do you know your program behaves as intended?* You've probably run it once or twice to make sure it does *something*, and maybe you've even tested it out with a few different inputs. But we can do better.

Especially when writing functions that perform a specific task, it is common to create **test cases** to call the function with specific inputs to ensure that the function works correctly and returns the "right" thing. This approach to testing is called **unit testing** because of the focus on testing individual functional units (i.e., the functions!) in a program.

You can think of testing your program in this way as something of an *experiment*. First, you decide on the inputs (parameters) you want to pass to your function. The output you expect from the function is basically a hypothesis which can easily be tested by running the function with your chosen input: if it produces the expected output, then your hypothesis was correct. If not, then there is probably something wrong with the function. (There may also be something wrong with the output you expected — for this reason you need to be very careful when devising tests!)

There is a built-in function called `assert` that can help with developing unit tests to ensure a function works as expected. The `assert` function takes a Boolean expression as a parameter. If the expression evaluates to `False`, the `assert` function will cause your program to crash. This is a good thing! The crash lets you know that something is wrong and needs to be fixed! If the expression in the `assert` function call evaluates to `True`, essentially nothing happens — the next line in the program will be executed.

Here's an example. The following function is supposed to take one string as a parameter, and count up and return the number of upper- and lower-case `'A'`s in the string. It has two bugs. Before you read on, see if you can figure out what they are.

```
# function should count up all the lower- and upper-case
# A's in a string and return the count.
def count_As(mystring):
    count = 0
    for char in mystring:
        if char == 'a':
            count += 1
```

Let's think of four test cases for our unit test of this function:

- If we call `count_As` with `'xyz'` (or an empty string), it should return 0.
- If we call `count_As` with `'abc'`, it should return 1.
- If we call `count_As` with `'ABC'`, it should also return 1.
- If we call `count_As` with `'Abracadabra'`, it should return 5.

We can construct calls to `assert` by including a Boolean expression that calls the function, and compares the return value to the expected output. In the `assert` calls, we are making assertions (duh!) about what the output should be:

```
def unit_tests():
    # three test cases using three different
    # strings to test whether the count_As
    # function works correctly
    assert(count_As("xyz") == 0)
    assert(count_As("abc") == 1)
    assert(count_As("ABC") == 1)
    assert(count_As("Abracadabra") == 5)

unit_tests()
```

When we run this program, we'll first call `unit_tests`, then we'll call each of the `assert` statements, in order. Because of the bugs in `count_As`, we'll crash on the first `assert` call:

```
Traceback (most recent call last):
  File "test2.py", line 16, in <module>
    unit_tests()
  File "test2.py", line 11, in unit_tests
    assert(count_As("xyz") == 0)
AssertionError
```

We see from the function stack traceback that the program died on line 11, which was the first call to `assert`. The program crashes because the return value of calling `count_As("xyz")` is not 0 (although it should be!) If we look carefully at the `count_As` function, we'll see one problem: there's no `return` statement! The function currently *always* returns `None`. Easy to fix:

```
def count_As(mystring):
    count = 0
    for char in mystring:
        if char == 'a':
            count += 1
    return count # added a return statement!
```

When we run the program now, we hit another `AssertionError`:

```
Traceback (most recent call last):
  File "test2.py", line 17, in <module>
    unit_tests()
  File "test2.py", line 14, in unit_tests
    assert(count_As("ABC") == 1)
AssertionError
```

Now, the program crashes on the `assert` on line 14. If we carefully examine the code (and perhaps add a `print` statement or two to help us to see what's going on in the function), we can see that we're only counting lower case `'a'`s, not upper case. Once we fix that problem, all the Boolean expressions in the `assert` calls will evaluate to `True`, and the program will finish without crashing. This will indicate that all our tests passed successfully.

Interestingly, an increasingly common practice within the software industry is to specify a set of test cases *before* writing a function. The idea is that the activity of specifying a set of test cases helps to clarify what a function should

do. Once the test cases are specified, the function can be written. Once all the test cases successfully pass, the function is done.

The hard part of testing a program is figuring out what a good set of test cases should be. Here are some rules of thumb:

- Pick one or two "normal" inputs and expected outputs. Think of parameters you expect to be commonly passed to the function and ensure that the function works for those parameters.
- Think about any "corner cases" — parameters that are just outside any "normal" or expected values. For example, if you usually expect to get the integers 1-10 as parameters to a function, write a tests for the valid bounds (1 and 10) as well as values just outside those bounds (0 and 11).
- Think deviously. What sorts of inputs might cause problems for a function? For example, if a function expects a string as input, what happens if an empty string (`""`) gets passed in?

You can also read more about unit testing on Wikipedia: http://en.wikipedia.org/wiki/Unit_testing.

5.10 Glossary

function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments 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.

function object: A value created by a function definition. The name of the function is a variable that refers to a function object.

function header: The first line of a function definition.

function body: The sequence of statements inside a function definition.

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

function call: A statement that executes a function. It consists of the function name followed by an argument list.

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

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

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

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

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

frame: A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

traceback: A list of the functions that are executing, printed when an exception occurs.

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

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

`None`: A special value returned by functions that have no return statement or a return statement without an argument.

test case: A set of parameters (inputs) and expected outputs for a function that can test whether the function behaves as expected.

unit testing: The idea of testing smaller pieces of a program, like a function, rather than testing the whole program at once.

assertion: A propositional statement that you expect to be `True` at some point in a program. The built-in `assert` function can be used to test Boolean propositional statements.

5.11 Exercises

1. Fix the last bug in the `count_As` function. Can you think of any additional test cases that should be added for this function?
2. Write a function named `compare_ab` that takes one string as parameters, and counts the occurrences of `'a'`s and `'b'`s in the string. The function should return `True` if the number of `'a'`s and `'b'`s is the same, and `False` otherwise. Think of a set of test cases for this function, and write them.
3. Write a function named `right_justify` that takes a string named `s` as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 60 of the display.

   ```
   >>> right_justify('allen')
                                                   allen
   ```

4. Write a function called `is_leap` that takes a year value as a parameter, and returns `True` if the year is a leap year or `False` if it is not. Refer to one of the exercises from the last chapter for the definition of a leap year.
5. A function object is a value you can assign to a variable or pass as an argument. For example, `do_twice` is a function that takes a function object as an argument and calls it twice:

   ```
   def do_twice(f):
       f()
       f()
   ```

   Here's an example that uses `do_twice` to call a function named `print_spam` twice.

   ```
   def print_spam():
       print 'spam'

   do_twice(print_spam)
   ```

   (a) Type this example into a script and test it.

   (b) Modify `do_twice` so that it takes two arguments, a function object and a value, and calls the function twice, passing the value as an argument.

   (c) Write a more general version of `print_spam`, called `print_twice`, that takes a string as a parameter and prints it twice.

   (d) Use the modified version of `do_twice` to call `print_twice` twice, passing `'spam'` as an argument.

   (e) Define a new function called `do_four` that takes a function object and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.
6. This exercise [2] can be done using only the statements and other features we have learned so far.

   Write a function that draws a grid like the following:

[2] Based on an exercise in Oualline, *Practical C Programming, Third Edition*, O'Reilly (1997)

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

Hint: to print more than one value on a line, you can print a comma-separated sequence:

```
print '+', '-'
```

If the sequence ends with a comma, Python leaves the line unfinished, so the value printed next appears on the same line.

```
print '+',
print '-'
```

The output of these statements is `'+ -'`.

A `print` statement all by itself ends the current line and goes to the next line.

7. Use the previous function to draw a similar grid with four rows and four columns.
8. Write a function that takes one integer named `size` as a parameter and prints an equilateral triangle composed of asterisks of length `size`. For example, the call `make_triangle(4)` should result in the following triangle printed:

```
   *
  * *
 * * *
* * * *
```

9. Draw a stack diagram for the following program. What does the program print?

```
def b(z):
    prod = a(z, z)
    print z, prod
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    sum = x + y + z
    pow = b(sum)**2
    return pow

x = 1
y = x + 1
print c(x, y+3, x+y)
```

10. Fermat's Last Theorem says that there are no integers a, b, and c such that:

$$a^n + b^n = c^n$$

for any values of n greater than 2.

(a) Write a function named `check_fermat` that takes four parameters—`a`, `b`, `c` and `n`—and that checks to see if Fermat's theorem holds. If n is greater than 2 and it turns out to be true that

$$a^n + b^n = c^n$$

the function should return `True`. Otherwise, the function should return `False`.

(b) Write a function that prompts the user to input values for `a`, `b`, `c` and `n`, converts them to integers, and uses `check_fermat` to check whether they violate Fermat's theorem. If the result of calling `check_fermat` is `False`, this function should print "Holy smokes, Fermat was wrong!". Otherwise, it should print "No, that doesn't work."

11. If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, it is clear that you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can.

(a) Write a function named `is_triangle` that takes three integers as arguments, and that returns `True` or `False`, depending on whether you can or cannot form a triangle from sticks with the given lengths.

(b) Write a function that prompts the user to input three stick lengths, converts them to integers, and uses `is_triangle` to check whether sticks with the given lengths can form a triangle.

CHAPTER 6

Program design

6.1 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to test and debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Solving a problem by writing and composing functions also lends itself well to a fundamental problem solving tactic: **divide and conquer**. The basic idea is to break down a problem into smaller, and more easily solved problems, until each subproblem can be solved directly. Many problems can be complex and difficult to try to solve as a whole, but if they are broken down into smaller subtasks, hopefully the subtasks are more manageable.

The main challenge with a divide-and-conquer approach to problem solving is *how to decompose the problem*: it isn't always apparent how to break a problem down into smaller components. As we work through more challenging problems, we will gain practice with this technique and learn different strategies for applying it.

6.2 Incremental development

Assuming you've figured out how to break down a problem into smaller subtasks, you are still faced with the challenge of writing larger and more complex functions. As a result, you might find yourself spending more time debugging.

To deal with writing increasingly complex programs, you might want to try a process called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the Pythagorean theorem, the distance is:

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

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

In this case, the inputs are two points, which you can represent using four numbers. The return value is the distance, which is a floating-point value.

Already you can write an outline of the function:

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

Obviously, this version doesn't compute distances; it always returns zero. But it is syntactically correct, and it runs, which means that you can test it before you make it more complicated.

To test the new function, call it with sample arguments:

```
>>> distance(1, 2, 4, 6)
0.0
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding code to the body. A reasonable next step is to find the differences $x_2 - x_1$ and $y_2 - y_1$. The next version stores those values in temporary variables and prints them.

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

If the function is working, it should display `'dx is 3'` and `'dy is 4'`. If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of `dx` and `dy`:

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

Again, you would run the program at this stage and check the output (which should be 25). Finally, you can use `math.sqrt` to compute and return the result:

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

If that works correctly, you are done. (Even better, you could construct additional test cases to verify that the function *really* works.) Otherwise, you might want to print the value of `result` before the return statement.

The final version of the function doesn't display anything when it runs; it only returns a value. The `print` statements we wrote are useful for debugging, but once you get the function working, you should remove them. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product.

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, incremental development 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 should have a good idea where it is.
2. Use temporary variables to hold intermediate values so you can display 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.

   Example:

   (a) 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 arguments. Record each stage of the development process as you go.

6.3 Turtles

In this section, we'll use the `turtle` module which is built in to Python as a way to get more practice with program design concepts. The `turtle` module provides a fairly simple way to draw on the screen. Here is an example to get started:

```
import turtle

def main():
    turtle.forward(100)  # move forward 100 units
    turtle.left(90)      # turn left 90 degrees

    turtle.forward(100)  # forward 100 units
    turtle.left(90)      # left 90 degrees

    turtle.forward(100)  # forward 100 units
    turtle.left(90)      # left 90 degrees

    turtle.forward(100)  # forward 100 units
    turtle.left(90)

    turtle.done()        # all done!

main()
```

The following screen shot shows the result of running the program. We first import the `turtle` module (which is built in to all Python versions). After that, we have a `main` function inside which we include all our turtle drawing statements. The `forward` function takes one parameter, which is the number of units to move forward. The `left` function takes a number of degrees to turn to the left. A turtle starts out in the middle of the screen, facing to the right. If you carefully read the above program, you'll see that we have the turtle draw each side of a square. At the end, the turtle is left facing directly right again.

The `turtle` module contains other functions to steer the turtle around the screen, including `backward`, `right`, `setposition`, and `setheading`. The Python `turtle` documentation has all the details on these functions: http://docs.python.org/library/turtle.html.

Also, each turtle is holding a pen, which is either down or up; if the pen is down, the turtle leaves a trail when it moves. The functions `penup` and `pendown` can control whether the pen is up or down. There is also a `pencolor` function

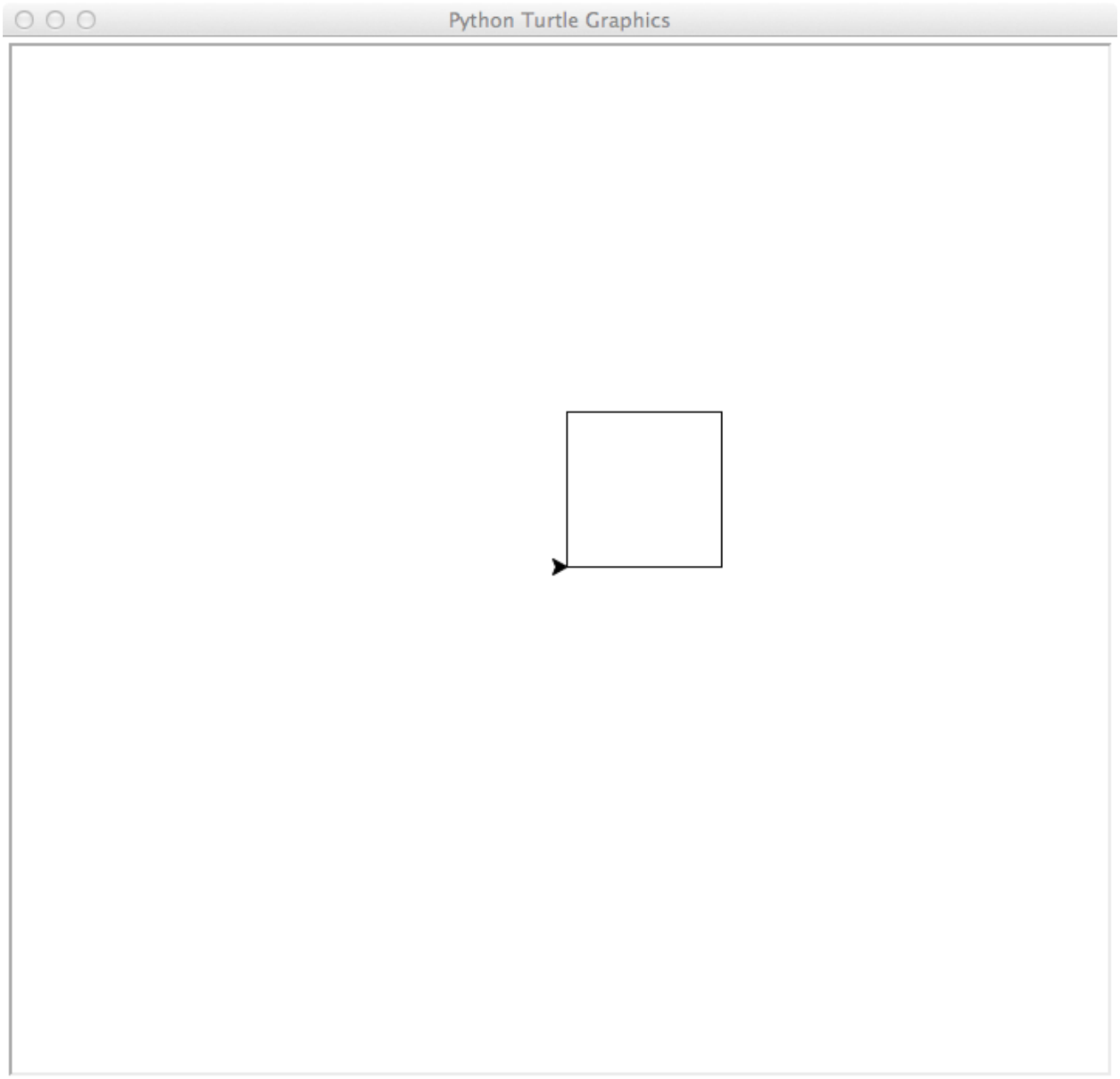


Figure 6.1: Turtle window after running the example program

that controls the color of the pen.

At the end of any `turtle` program, you should always call the `done` function. If you do not call this function, you may need to restart IDLE after running a program that uses `turtle`. (Different operating systems behave differently in this regard. Especially on Windows systems, you should remember to call `turtle.done()` at the end of a turtle drawing program.)

6.4 Exercise

1. See if you can simplify the above program by using a `for` loop to draw each side of the square.

The following is a series of exercises using `turtle`. They are meant to be fun, but they have a point, too. While you are working on them, think about what the point is.

The sections that immediately follow have solutions to the exercises, so don't look until you have finished (or at least tried).

1. Write a function called `square` that uses `turtle` to draw a square.
2. Add a parameter, named `length`, to `square`. Modify the function so length of the sides is `length`, and then modify the function call to provide an argument for the length. Run the program again. Test your program with a range of values for `length`.
3. Make a copy of `square` and change the name to `polygon`. Add another parameter named `n` and modify the body so it draws an n-sided regular polygon.

   Hint: instead of passing 90 to the `left` or `right` function for turning the turtle, you'll need to specify a different value. As another hint, the exterior angles of an n-sided regular polygon are $360.0/n$ degrees.
4. Write a function called `circle` that takes a radius, `r`, as a parameter and that draws an approximate circle by invoking `polygon` with an appropriate length and number of sides. Test your function with a range of values of `r`.

   Hint: figure out the circumference of the circle and make sure that `length * n = circumference`.

   Another hint: if the turtle drawing is too slow for your taste, you can call `turtle.speed('fastest')`.
5. Make a more general version of `circle` called `arc` that takes an additional parameter `angle`, which determines what fraction of a circle to draw. `angle` is in units of degrees, so when `angle=360`, `arc` should draw a complete circle.

6.5 Encapsulation

The first exercise asks you to put your square-drawing code into a function definition and then call the function, passing the turtle as a parameter. Here is a solution:

```
import turtle

def square():
    for i in range(4):
        turtle.forward(100)
        turtle.left(90)

square()
turtle.done()
```

The innermost statements, `forward` and `left` are indented twice to show that they are inside the `for` loop, which is inside the function definition. The next line, `square()`, is flush with the left margin, so that is the end of both the `for` loop and the function definition.

The `for` loop above is a bit odd in the sense that we never use the variable `i` inside the statement body. This isn't uncommon in situations in which we want a statement body to be repeated a specific number of times, but we don't necessarily have to keep track of which iteration we're on.

Wrapping a piece of code up in a function is called **encapsulation**. One of the benefits of encapsulation is that it attaches a name to the code, which serves as a kind of documentation. Another advantage is that if you re-use the code, it is more concise to call a function twice than to copy and paste the body!

6.6 Generalization

The next step is to add a `length` parameter to `square`. Here is a solution:

```
import turtle

def square(length):
    for i in range(4):
        turtle.forward(length)
        turtle.left(90)

square(100)
turtle.done()
```

Adding a parameter to a function is called **generalization** because it makes the function more general: in the previous version, the square is always the same size; in this version it can be any size.

The next step is also a generalization. Instead of drawing squares, `polygon` draws regular polygons with any number of sides. Here is a solution:

```
import turtle

def polygon(n, length):
    angle = 360.0 / n
    for i in range(n):
        turtle.forward(length)
        turtle.left(angle)

polygon(7, 70)
turtle.done()
```

This draws a 7-sided polygon with side length 70. If you have more than a few numeric arguments, it is easy to forget what they are, or what order they should be in. It is legal, and sometimes helpful, to include the names of the parameters in the argument list:

```
polygon(n=7, length=70)
```

These are called **keyword arguments** because they include the parameter names as "keywords" (not to be confused with Python keywords like `for` and `def`).

This syntax makes the program more readable. It is also a reminder about how arguments and parameters work: when you call a function, the arguments are assigned to the parameters.

6.7 Interface design

The next step is to write `circle`, which takes a radius, `r`, as a parameter. Here is a simple solution that uses `polygon` to draw a 50-sided polygon:

```
def circle(r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(n, length)
```

The first line computes the circumference of a circle with radius `r` using the formula $2\pi r$. Since we use `math.pi`, we have to import `math`. Remember that by convention, `import` statements should be put at the beginning of the script.

`n` is the number of line segments in our approximation of a circle, so `length` is the length of each segment. Thus, `polygon` draws a 50-sides polygon that approximates a circle with radius `r`.

One limitation of this solution is that `n` is a constant, which means that for very big circles, the line segments are too long, and for small circles, we waste time drawing very small segments. One solution would be to generalize the function by taking `n` as a parameter. This would give the user (whoever calls `circle`) more control, but the interface would be less clean.

The **interface** of a function is a summary of how it is used: what are the parameters? What does the function do? And what is the return value? An interface is "clean" if it is "as simple as possible, but not simpler." (Einstein)

In this example, `r` belongs in the interface because it specifies the circle to be drawn. `n` is less appropriate because it pertains to the details of *how* the circle should be rendered.

Rather than clutter up the interface, it is better to choose an appropriate value of `n` depending on `circumference`:

```
def circle(r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 1
    length = circumference / n
    polygon(n, length)
```

Now the number of segments is (approximately) `circumference/3`, so the length of each segment is (approximately) 3, which is small enough that the circles look good, but big enough to be efficient, and appropriate for any size circle.

6.8 Refactoring

When we wrote `circle`, we were able to re-use `polygon` because a many-sided polygon is a good approximation of a circle. But `arc` is not as cooperative; we can't use `polygon` or `circle` to draw an arc.

One alternative is to start with a copy of `polygon` and transform it into `arc`. The result might look like this:

```
def arc(r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n

    for i in range(n):
        turtle.forward(step_length)
        turtle.left(step_angle)
```

The second half of this function looks like `polygon`, but we can't re-use `polygon` without changing the interface. We could generalize `polygon` to take an angle as a third argument, but then `polygon` would no longer be an appropriate name! Instead, let's call the more general function `polyline`:

```
def polyline(n, length, angle):
    for i in range(n):
        turtle.forward(length)
        turtle.forward(angle)
```

Now we can rewrite `polygon` and `arc` to use `polyline`:

```
def polygon(n, length):
    angle = 360.0 / n
    polyline(n, length, angle)

def arc(r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(n, step_length, step_angle)
```

Finally, we can rewrite `circle` to use `arc`:

```
def circle(r):
    arc(r, 360)
```

This process—rearranging a program to improve function interfaces and facilitate code re-use—is called **refactoring**. In this case, we noticed that there was similar code in `arc` and `polygon`, so we "factored it out" into `polyline`.

If we had planned ahead, we might have written `polyline` first and avoided refactoring, but often you don't know enough at the beginning of a project to design all the interfaces. Once you start coding, you understand the problem better. Sometimes refactoring is a sign that you have learned something.

Here's the full set of code we wrote:

```
import turtle
import math

def polyline(n, length, angle):
    for i in range(n):
        turtle.forward(length)
        turtle.left(angle)

def polygon(n, length):
    angle = 360.0 / n
    polyline(n, length, angle)

def arc(r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(n, step_length, step_angle)

def circle(r):
    arc(r, 360)
```

6.9 A development plan

A **development plan** is a process for writing programs. The process we used in this case study is "encapsulation and generalization." The steps of this process are:

1. Start by writing a small program with no function definitions.
2. Once you get the program working, encapsulate it in a function and give it a name.
3. Generalize the function by adding appropriate parameters.
4. Repeat steps 1–3 until you have a set of working functions. Copy and paste working code to avoid retyping (and re-debugging).
5. Look for opportunities to improve the program by refactoring. For example, if you have similar code in several places, consider factoring it into an appropriately general function.

This process has some drawbacks, but it can be useful if you don't know ahead of time how to divide the program into functions. This approach lets you design as you go along.

6.10 docstring

A **docstring** is a string at the beginning of a function that explains the interface ("doc" is short for "documentation"). Here is an example:

```
def polyline(length, n, angle):
    """Draw n line segments with the given length and
    angle (in degrees) between them.
    """
    for i in range(n):
        turtle.forward(length)
        turtle.left(angle)
```

This docstring is a triple-quoted string, also known as a multiline string because the triple quotes allow the string to span more than one line.

It is terse, but it contains the essential information someone would need to use this function. It explains concisely what the function does (without getting into the details of how it does it). It explains what effect each parameter has on the behavior of the function and what type each parameter should be (if it is not obvious).

Writing this kind of documentation is an important part of interface design. A well-designed interface should be simple to explain; if you are having a hard time explaining one of your functions, that might be a sign that the interface could be improved.

6.11 Debugging

An interface is like a contract between a function and a caller. The caller agrees to provide certain parameters and the function agrees to do certain work.

For example, `polyline` requires three arguments. The first has to be a number, and it should probably be positive, although it turns out that the function works even if it isn't. The second argument should be an integer; `range` complains otherwise (depending on which version of Python you are running). The third has to be a number, which is understood to be in degrees.

These requirements are called **preconditions** because they are supposed to be true before the function starts executing. Conversely, conditions at the end of the function are **postconditions**. Postconditions include the intended effect of the function (like drawing line segments) and any side effects (like moving the turtle).

Preconditions are the responsibility of the caller. If the caller violates a (properly documented!) precondition and the function doesn't work correctly, the bug is in the caller, not the function.

Note that the `assert` function described in the last chapter can be incredibly helpful for verifying pre- or post-conditions.

6.12 Glossary

divide and conquer: A problem solving strategy that proceeds by breaking down a problem into smaller and smaller subtasks, until a subtask can be solved directly.

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

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

encapsulation: The process of transforming a sequence of statements into a function definition.

generalization: The process of replacing something unnecessarily specific (like a number) with something appropriately general (like a variable or parameter).

keyword argument: An argument that includes the name of the parameter as a "keyword."

interface: A description of how to use a function, including the name and descriptions of the arguments and return value.

refactoring: The process of modifying a working program to improve function interfaces and other qualities of the code.

development plan: A process for writing programs.

docstring: A string that appears in a function definition to document the function's interface.

precondition: A requirement that should be satisfied by the caller before a function starts.

postcondition: A requirement that should be satisfied by the function before it ends.

6.13 Exercises

1. Write an appropriately general set of functions that can draw flowers like this:

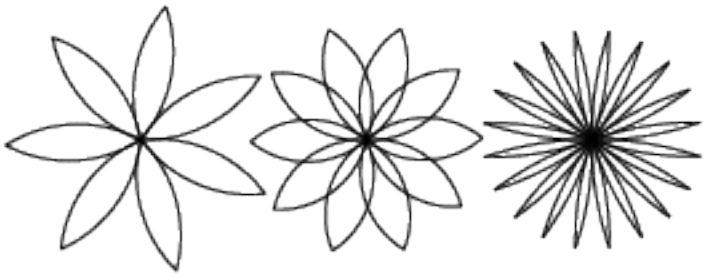


Figure 6.2: Example flowers to draw with turtle graphics.

2. Write an appropriately general set of functions that can draw shapes like this:

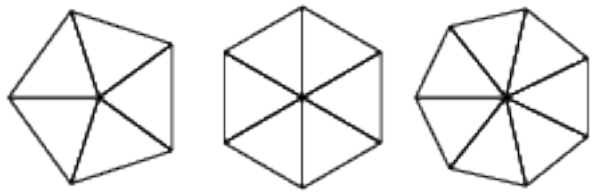


Figure 6.3: Example shapes to draw with turtle graphics.

3. The letters of the alphabet can be constructed from a moderate number of basic elements, like vertical and horizontal lines and a few curves. Design a font that can be drawn with a minimal number of basic elements and then write functions that draw letters of the alphabet.

   You should write one function for each letter, with names `draw_a`, `draw_b`, etc., and put your functions in a file named `letters.py`.

CHAPTER 7

Iteration

In this chapter, we'll learn about a way to repeat (loop) over a set of statements that is more general than the `for` loop we have already encountered. First, a slight detour to learn a bit more about lists.

7.1 An introduction to lists

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be *any* type. The values in a list are called **elements** or sometimes **items**.

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

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. (Mmmm... crunchy frog.) The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

A list that contains no elements is called an empty list; you can create one with empty brackets, `[]`.

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

7.1.1 Lists are mutable

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

```
>>> print cheeses[0]
Cheddar
```

Unlike strings, lists are *mutable*. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be modified through assignment.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

The one-eth element of `numbers`, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a **mapping**; each index "maps to" one of the elements. Here is a state diagram showing `cheeses`, `numbers` and `empty`:

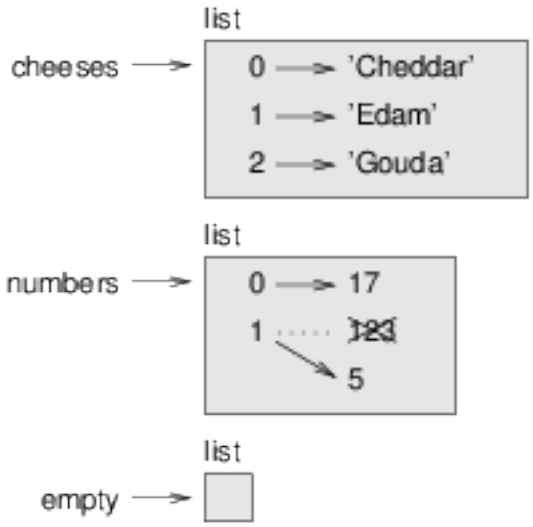


Figure 7.1: Using lists to make "mappings".

Lists are represented by boxes with the word "list" outside and the elements of the list inside. `cheeses` refers to a list with three elements indexed 0, 1 and 2. `numbers` contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. `empty` refers to a list with no elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

The `in` operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

7.1.2 Traversing a list

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

```
for cheese in cheeses:
    print cheese
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to $n - 1$, where n is the length of the list. Each time through the loop `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A `for` loop over an empty list never executes the body:

```
for x in []:
    print 'This never happens.'
```

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

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

7.1.3 `range` revisited

Earlier, we learned to use the `range` function to loop over the indices of a string:

```
# count the lower case e's in a string
count = 0
s = "that's cheesy"
for index in range(len(s)):
    if s[index] == 'e':
        count += 1
```

We learned that `range`, when given an integer parameter, will cause the `for` loop iterator variable (`index`, above) to "take on" the values 0, 1, 2, ... `len(s)-1`.

Besides accepting one parameter, `range` can accept either 2 or 3 parameters, giving us more flexibility in how to generate a list of numbers. When called with two parameters, `range(i, j)` returns the list of integers `[i, i+1, i+2, ..., j-1]`. When called with three parameters, `range(i, j, k)` the third value represents a `step` to increment or decrement by when creating the sequence. It is, by default, 1, but can be any integral value. Here are some examples with `range`:

```
>>> range(4)
[0, 1, 2, 3]
>>> range(1,4)
[1, 2, 3]
>>> range(0,4,2)
[0, 2]
>>> range(0,4,3)
[0, 3]
```

7.1.4 List operations

The + operator concatenates lists, similar to string concatenation:

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

In the same way, 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]
```

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

There are four useful built-in functions relevant to lists. We've already seen `len`, which returns the length of a sequence. There are also `min`, `max`, and `sum`, which return the minimum and maximum values in a list, and the sum of a list of integers, respectively. For example:

```
>>> mylist = [1,3,6,42]
>>> len(mylist)
4
>>> min(mylist)
1
>>> max(mylist)
42
>>> sum(mylist)
52
```

7.2 The `while` statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

We have seen several programs that use the `for` statement to perform repetition, which is also called **iteration**. Because iteration is so common, Python provides several language features to make it easier. One is, of course, the `for` statement, and another is the `while` statement.

Let's say we're obsessed with rockets blasting off to the moon, and that we want a function that prints a "countdown" sequence. Here is how we might do that with a `while` statement:

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

You can almost read the `while` statement as if it were English:

> "While `n` is greater than 0, display the value of `n` and then reduce the value of `n` by 1. When you get to 0, display the word `Blastoff!`".

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

1. Evaluate the condition, yielding `True` or `False`.

2. If the condition is `False`, exit the `while` statement and continue execution at the next statement.
3. If the condition is `True`, execute the body and then go back to step 1 (thus making a **loop**).

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**.

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

```
def sequence(n):
    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 makes the condition false.

Each time through the loop, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, `n` is divided by 2. If it is odd, the value of `n` is replaced with `n*3+1`. For example, if the argument passed to `sequence` is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program terminates. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

The hard question is whether we can prove that this program terminates for *all positive values* of `n`. So far [1], no one has been able to prove it *or* disprove it!

7.3 break

Sometimes you don't know whether to end a loop until you get half way through the body. In that case you can use the `break` statement to "jump" out of the loop.

For example, suppose you want to take input from the user until they type `done`. You could write:

```
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line

print 'Done!'
```

The loop condition is `True`, which is always true, so the loop runs until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types `done`, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> not done
not done
> done
Done!
```

[1] See http://wikipedia.org/wiki/Collatz_conjecture.

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens.").

7.4 Square roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula:

$$y = \frac{x + \frac{a}{x}}{2}$$

For example, if a is 4 and x is 3:

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print y
2.16666666667
```

Which is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00641025641
```

After a few more updates, the estimate is almost exact:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00000000003
```

In general we don't know ahead of time how many steps it will take to get to the right answer, but we know when we get there because the estimate stops changing:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
```

When `y == x`, we can stop. Here is a loop that starts with an initial estimate, `x`, and improves it until it stops changing:

```
while True:
    print x
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

For most values of `a` this works fine, but in general it is dangerous to test `float` equality. Floating-point values are only approximately right: most rational numbers, like $1/3$, and irrational numbers, like $\sqrt{2}$, can't be represented exactly with a `float`.

Rather than checking whether `x` and `y` are exactly equal, it is safer to use the built-in function `abs` to compute the absolute value, or magnitude, of the difference between them:

```
if abs(y-x) < epsilon:
    break
```

Where `epsilon` has a value like `0.0000001` that determines how close is close enough.

Example:

1. Encapsulate this loop in a function called `square_root` that takes `a` as a parameter, chooses a reasonable value of `x`, and returns an estimate of the square root of `a`.

7.5 Algorithms

Newton's method is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots).

It is not easy to define an algorithm. It might help to start with 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.

The process of designing algorithms is interesting, intellectually challenging, and a central part of what we call *program design*. Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

7.6 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more place for bugs to hide.

One way to cut your debugging time is "debugging by bisection". For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a `print` statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you cut in half the number of lines that have to be searched. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the "middle of the program" is and not always possible to check it. It doesn't make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

7.7 Glossary

initialization: An assignment that gives an initial value to a variable that will be updated.

increment: An update that increases the value of a variable (often by one).

decrement: An update that decreases the value of a variable.

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

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

7.8 Exercises

1. To test the square root algorithm in this chapter, you could compare it with `math.sqrt`. Write a function named `test_square_root` that prints a table like this:

a	**square_root(a)**	**math.sqrt(a)**	**difference**
1.0	1.0	1.0	0.0
2.0	1.41421356237	1.41421356237	2.22044604925e-16
3.0	1.73205080757	1.73205080757	0.0
4.0	2.0	2.0	0.0
5.0	2.2360679775	2.2360679775	0.0
6.0	2.44948974278	2.44948974278	0.0
7.0	2.64575131106	2.64575131106	0.0
8.0	2.82842712475	2.82842712475	4.4408920985e-16
9.0	3.0	3.0	0.0

   The first column is a number, a; the second column is the square root of a computed with the `square_root` function; the third column is the square root computed by `math.sqrt`; the fourth column is the absolute value of the difference between the two estimates.

2. The built-in function `eval` takes a string and evaluates it using the Python interpreter. For example:

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<type 'float'>
```

   Write a function called `eval_loop` that iteratively prompts the user, takes the resulting input and evaluates it using `eval`, and prints the result.

It should continue until the user enters `'done'`, and then return the value of the last expression it evaluated.

3. The brilliant mathematician Srinivasa Ramanujan found an infinite series [2] that can be used to generate a numerical approximation of π:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Write a function called `estimate_pi` that uses this formula to compute and return an estimate of π. It should use a `while` loop to compute terms of the summation until the last term is smaller than `1e-15` (which is Python notation for 10^{-15}). You can check the result by comparing it to `math.pi`.

[2] See http://wikipedia.org/wiki/Pi

CHAPTER 8

Strings

In this chapter, we dive into a bit more depth on strings.

8.1 Traversal with a `while` loop

There are a number of situations in which a string must be processed one character at a time. We've already done this with `for` loops, but `while` loops give us some additional flexibility. Often, we 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**. For example:

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

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

Examples

1. Write a function that takes a string as an argument and displays the letters backward, one per line.
2. Write a function that takes a string as an argument and displays the letters backward, all on the same line. (A space between letters is ok.)
3. Write a function that takes a string as a parameter and returns the number of `'x'`s that appear before the first `'z'` in the string. For example, calling the function with the string `'xaxxyz!x?'` should yield the count 3, since there are 3 `'x'`s that appear before the `'z'` in the string.

The following example shows how to use concatenation (string addition) and a `for` loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
```

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

The output is:

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

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

Example:

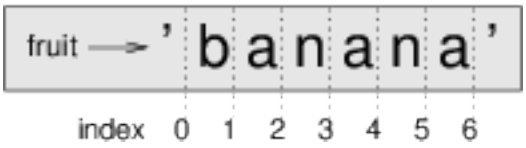
1. Modify the program to fix this error.
2. Modify the program to use a `while` loop instead of a `for` loop.

8.2 String slices

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

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:12]
Python
```

The operator `[n:m]` returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters, as in the following diagram:



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

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

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Another way to slice a string is to use *three* indices. The third value is referred to as the `step`:

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

The three slice indices work similarly to the three arguments to the `range` function. In fact, you can think of the three slice parameters as being used in a `range` function call to *generate* the indices of values to be extracted ("sliced out") from the string. `range(0,5,2)` would give the list `[0, 2, 4]`, so the slice yields a string composed of the characters at indices 0, 2, and 4.

Example:

1. Given that `fruit` is a string, what does `fruit[:]` mean?
2. Can you construct a string slice that will return a reversed copy of the string?

8.3 Strings are immutable

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

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an **object** is the same thing as a value, but we will refine that definition later. An **item** is one of the values in a sequence.

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

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

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

8.4 Searching

What does the following function do? Read the function carefully before moving on. If it helps, you can trace the operation of two different calls to this function: `find('magic', 'i')` and `find('magic', 'z')`.

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            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 appears. If the character is not found, the function returns `-1`.

This is the first example we have seen of a `return` statement inside a loop. If `word[index] == letter`, the function breaks out of the loop and returns immediately.

If the character doesn't appear in the string, the program exits the loop normally and returns `-1`.

This pattern of computation—traversing a sequence and returning when we find what we are looking for—is called a **search**.

Example:

1. Modify `find` so that it has a third parameter, the index in `word` where it should start looking.
2. Write a function named `findall` that takes a character to search for and a string, and returns a list of indices where the character is found in the string.

8.5 Looping and counting

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

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print count
```

This program demonstrates another pattern of computation called a **counter**. The variable `count` is initialized to 0 and then incremented each time an `a` is found. When the loop exits, `count` contains the result—the total number of `a`'s.

Examples:

1. Encapsulate this code in a function named `count`, and generalize it so that it accepts the string and the letter as arguments.
2. Rewrite this function so that instead of traversing the string, it uses the three-parameter version of `find` from the previous section.
3. Rewrite the function so that instead of passing a single character as a parameter, another string can be passed to the function. Try to generalize the function so that it works for any length of substring.

8.6 `string` methods

A **method** is similar to a function —it takes arguments and returns a value—but the syntax is different. For example, the method `upper` takes a string and returns a new string with all uppercase letters:

Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no argument.

A method call is called an **invocation**; in this case, we would say that we are invoking `upper` on the `word`.

As it turns out, there is a string method named `find` that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

Actually, the `find` method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

And as a third argument the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range from `1` to `2` (not including `2`).

Example:

1. There is a string method called `count` that is similar to the function in the previous exercise. Read the documentation of this method and write an invocation that counts the number of `a`s in `'banana'`.

There are quite a few string methods, and you'll probably want to take a look at the documentation: http://docs.python.org/library/stdtypes.html#string-methods. Below, we review several of the useful methods:

method	description
`upper`	Return an upper-cased copy of the string.
`lower`	Return a lower-cased copy of the string.
`capitalize`	Return a copy of the string with the first character
	capitalized.
`count(s)`	Return the number of non-overlapping occurrences
	of the substring `s` in the string.
`replace(old, new)`	Return a copy of the string with all occurrences of
	`old` replaced by `new`.
`strip`	Return a copy of the string with leading and trailing
	"whitespace" characters removed (spaces, tabs, and
	newline characters).
`split`	Return a list of words in the string, separating the
	string by any whitespace characters.

Note that several of the methods above can take optional parameters, which modify the behavior of the method. Refer to the Python documentation for details on the various string methods.

8.7 Character-numeric duality

Internal to a computer, *all* data are represented *numerically*: images, sounds, videos, strings, and characters. Sometimes it is useful to be able to process characters *numerically* instead of as single-character strings. Python includes

two built-in functions to help with this: `ord` and `chr`.

`ord(ch)` returns the numeric, or *ordinal* value of a character. `chr(n)` returns the character corresponding to a given number `n`. For example:

```
>>> ord('A')
65
>>> ord('B')
66
>>> ord('C')
67
>>> ord('a')
97
>>> ord('b')
98
>>> chr(99)
'c'
>>> chr(100)
'd'
```

As you can see above, upper case letters and lower case letters are each organized sequentially. Upper case letters start at the ordinal value 65, and lower case letters start at 97. Knowing these specific numbers is not important; it is useful to observe, however, that they're organized sequentially.

The mappings between characters and their numeric equivalents is defined by several standards. The most historically relevant one is the American Standard Code for Information Interchange, or ASCII: http://en.wikipedia.org/wiki/Ascii. Unfortunately, ASCII, as the name suggests, is United States (and English) centric and cannot accommodate character sets from other languages such as Chinese, Russian, or Korean. The Unicode standard was developed to accommodate international character sets. Unicode is beyond the scope of this class, but if you're interested, you can read more on Wikipedia: http://en.wikipedia.org/wiki/Unicode.

8.8 String comparison and ordering

As we've already seen, the relational operators work on strings. However, Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so `'Pineapple'` comes before `'banana'`.

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

8.9 Debugging

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return `True` if one of the words is the reverse of the other, but it contains two errors:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)
```

```
    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

The first `if` statement checks whether the words are the same length. If not, we can return `False` immediately and then, for the rest of the function, we can assume that the words are the same length.

`i` and `j` are indices: `i` traverses `word1` forward while `j` traverses `word2` backward. If we find two letters that don't match, we can return `False` immediately. If we get through the whole loop and all the letters match, we return `True`.

If we test this function with the words "pots" and "stop", we expect the return value `True`, but we get an IndexError:

```
>>> is_reverse('pots', 'stop')
...
  File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

For debugging this kind of error, my first move is to print the values of the indices immediately before the line where the error appears.

```
while j > 0:
    print i, j        # print here

    if word1[i] != word2[j]:
        return False
    i = i+1
    j = j-1
```

Now when I run the program again, I get more information:

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

The first time through the loop, the value of `j` is 4, which is out of range for the string `'pots'`. The index of the last character is 3, so the initial value for `j` should be `len(word2)-1`.

If I fix that error and run the program again, I get:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

This time we get the right answer, but it looks like the loop only ran three times, which is suspicious. To get a better idea of what is happening, it is useful to draw a state diagram. During the first iteration, the frame for `is_reverse` looks like this:

I took a little license by arranging the variables in the frame and adding dotted lines to show that the values of `i` and `j` indicate characters in `word1` and `word2`.

Example:

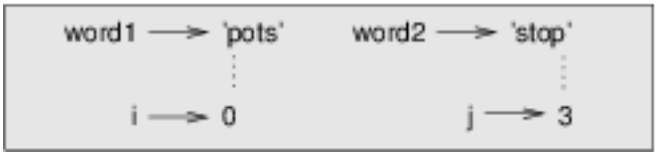


Figure 8.1: State diagram for `is_reverse` example.

1. Starting with this diagram, execute the program on paper, changing the values of `i` and `j` during each iteration. Find and fix the second error in this function.

8.10 Glossary

object: Something a variable can refer to. For now, you can use "object" and "value" interchangeably.

sequence: An ordered set; that is, a set of values where each value is identified by an integer index.

item: One of the values in a sequence.

index: An integer value used to select an item in a sequence, such as a character in a string.

slice: A part of a string specified by a range of indices.

empty string: A string with no characters and length 0, represented by two quotation marks.

immutable: The property of a sequence whose items cannot be assigned.

traverse: To iterate through the items in a sequence, performing a similar operation on each.

search: A pattern of traversal that stops when it finds what it is looking for.

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

method: A function that is associated with an object and called using dot notation.

invocation: A statement that calls a method.

8.11 Exercises

1. Write a function named `inboth` that takes two strings as parameters, and returns a list of all the characters that are contained in both strings. Write the function in a case-sensitive way.
2. Rewrite the `inboth` function to work in a case-insensitive way.
3. The following functions are all *intended* to check whether a string contains any lowercase letters, but at least some of them are wrong. For each function, describe what the function actually does (assuming that the parameter is a string).

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
```

```
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

4. ROT13 is a weak form of encryption that involves "rotating" each letter in a word by 13 places [1]. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so 'A' shifted by 3 is 'D' and 'Z' shifted by 1 is 'A'.

   Write a function called `rotate_word` that takes a string and an integer as parameters, and that returns a new string that contains the letters from the original string "rotated" by the given amount.

   For example, "cheer" rotated by 7 is "jolly" and "melon" rotated by -10 is "cubed".

5. Write a program that asks for a phrase, then computes and prints the number of words and number of characters in the phrase.

6. Write a program that asks for a phrase, then computes the number of upper and lower case letters, and prints the two counts.

[1] See http://wikipedia.org/wiki/ROT13.

CHAPTER 9

Recursion

9.1 Chicken, meet egg

It is legal for one function to call another; 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 things a program can do. A function that calls itself is **recursive**; the process is called **recursion**.

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:

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

If you saw that definition in the dictionary, you might be annoyed (in which case, you might want to check out "recursive acronyms"! http://en.wikipedia.org/wiki/Recursive_acronym). On the other hand, if you looked up the definition of the factorial function, denoted with the symbol !, you might get something like this:

- $0! = 1$
- $n! = n * (n-1)!$

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, 3! equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that `factorial` takes an integer:

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

If the argument happens to be 0, 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 call `factorial` with the value 3:

> Since 3 is not 0, we take the second branch and calculate the factorial of `n-1`...
>
> > Since 2 is not 0, we take the second branch and calculate the factorial of `n-1`...
> >
> > > Since 1 is not 0, we take the second branch and calculate the factorial of `n-1`...
> > >
> > > > Since 0 *is* 0, we take the first branch and return 1 without making any more recursive calls.
> > >
> > > The return value (1) is multiplied by n, which is 1, and the result is returned.
> >
> > The return value (1) is multiplied by n, which is 2, and the result is returned.
>
> The return value (2) is multiplied by n, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

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

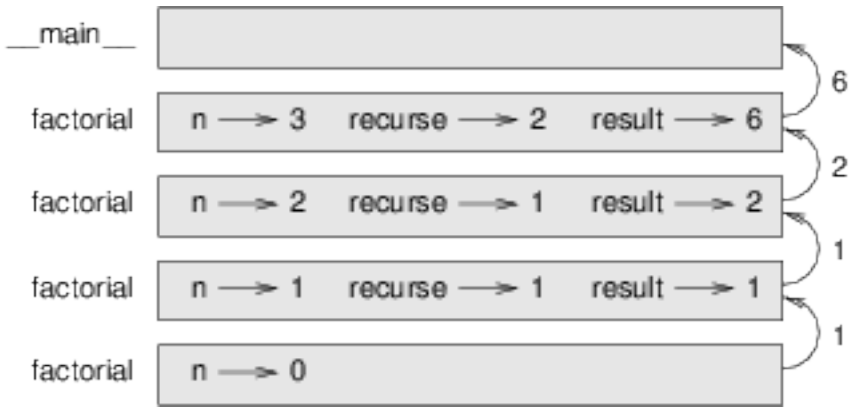


Figure 9.1: Stack diagram for `factorial(3)`

The four frames have different values for the parameter `n`. The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of `n` and `recurse`.

The bottom of the stack, where `n = 0`, is referred to as the **base case**. It does not make a recursive call, so there are no more stack frames. Also, in the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them does not execute.

9.2 Decrease and conquer

Recursion as a programming technique lends itself well to a problem solving strategy called **decrease and conquer**. The basic idea is to consider how a problem can be formulated in terms of "smaller" versions of itself. In the case of the `factorial` function, we directly use the mathematical definition to accomplish this:

```
factorial(n) = n * factorial(n-1)
```

The right-hand side of the statement *decreases* the problem size (`n`) by one, and recursively invokes the `factorial` function. Since this statement recursively reduces the problem size, we eventually reach the **base case** (i.e., `n == 0`), at which point the recursion stops.

The strategy discussed earlier, *divide and conquer* is also useful when thinking about solving problems recursively. If a problem can be divided into smaller, non-overlapping versions of itself, a recursive approach may be appropriate. In the searching and sorting chapters, we'll see an examples of two problems that fit this approach.

9.3 Infinite recursion

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

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

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1000 `recurse` frames on the stack!

9.4 Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become labyrinthine. An alternative approach is what might be considered 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 right result.

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

The same is true when you call one of your own functions. For example, *we previously wrote a function called ‘‘is_divisible‘ <#sec:booleanfn>‘_* that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—by examining the code and testing—we can use the function without looking at the body again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (yields the correct result) and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

Of course, it's a bit strange to assume that the function works correctly when you haven't finished writing it, but that's why it's called a leap of faith!

9.5 Two more examples

9.5.1 Factorial

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`. Similar to the `factorial` function, it follows a *decrease and conquer* approach. The `fibonacci` function has the following definition [1]:

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

Translated into Python, it looks like this:

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

Notice that in the last line, the "problem size" (`n`) is reduced by one or two. Eventually, we decrease the problem size to `n = 0` or `n = 1`. If you try to follow the flow of execution here, even for fairly small values of n, your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

9.5.2 Palindromes

Another problem that can solved with a recursive "reduce and conquer" approach is to determine whether a string is a palindrome or not. A string is a palindrome if it is spelled the same way backward and forward. For example, the following words are palindromes:

- racecar
- civic
- kayak
- rotator
- testset

The following phrase is also palindromic (assuming the punctuation is removed, and we convert all characters to the same case):

> A man, a plan, a canal, Panama!

This problem is a little bit different than the others we've seen, but the idea of reducing the problem to a smaller size still holds. First, we'll define a base case:

> A string of length 1 or less is a palindrome.

Nothing very controversial there, right? And it suggests an approach: try to reduce the size of the string until we have a string of length 1 or less.

[1] See http://wikipedia.org/wiki/Fibonacci_number.

The main bit of insight we need is to realize that for a string to be a palindrome, *the first and last characters must be the same*. If they are, we can slice off the first and last characters, thus reducing the problem size! We can then check if the remaining string is a palindrome — recursion! If the first and last characters aren't the same, there's no chance the string is a palindrome. Translating all that to a function:

```
def isPalindrome(s):
    # string of 1 or fewer characters
    # is necessarily a palindrome
    if len(s) <= 1:
        return True

    # if first and last characters are
    # the same, reduce the problem size,
    # and make recursive call
    elif s[0] == s[-1]:
        newstring = s[1:-1]
        return isPalindrome(newstring)

    # no chance that we've got a palindrome.
    # just return in abject failure.
    else:
        return False
```

9.6 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? There is a base case—when `n == 0`. But if `n` is not an integer, we can *miss* the base case and recurse forever.

In the first recursive call, the value of `n` is 0.5. In the next, it is -0.5. From there, it gets smaller (more negative), but it will never be 0.

We have two choices. We can try to generalize the `factorial` function to work with floating-point numbers, or we can make `factorial` check the type of its argument. The first option is called the gamma function [2] and it's a little beyond the scope of this book. So we'll go for the second.

We can use the built-in function `isinstance` to verify the type of the argument. While we're at it, we can also make sure the argument is positive:

```
def factorial (n):
    if not isinstance(n, int):
        print 'Factorial is only defined for integers.'
        return None
    elif n < 0:
        print 'Factorial is not defined for negative integers.'
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

[2] See http://wikipedia.org/wiki/Gamma_function.

The first base case handles nonintegers; the second catches negative integers. In both cases, the program prints an error message and returns `None` to indicate that something went wrong:

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is not defined for negative integers.
None
```

If we get past both checks, then we know that n is positive or zero, so we can prove that the recursion terminates.

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

9.7 A theoretical aside

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

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. For a more complete (and accurate) discussion of the Turing Thesis, I recommend Michael Sipser's book *Introduction to the Theory of Computation*.

9.8 Debugging

Adding print statements at the beginning and end of a function can help make the flow of execution more visible, especially when debugging recursive functions. For example, here is a version of `factorial` with print statements:

```
def factorial(n):
    space = ' ' * (4 * n)
    print space, 'factorial', n
    if n == 0:
        print space, 'returning 1'
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print space, 'returning', result
        return result
```

`space` is a string of space characters that controls the indentation of the output. Here is the result of `factorial(5)`:

```
                    factorial 5
                factorial 4
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
```

```
returning 1
    returning 2
        returning 6
            returning 24
                returning 120
```

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but a little bit of scaffolding can save a lot of debugging.

9.9 Glossary

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

base case: A conditional branch in a recursive function that does not make a recursive call.

infinite recursion: A recursion that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

9.10 Exercises

1. Write a function that takes a possibly empty list of integers as a parameter, and recursively computes and returns the sum of the list of numbers. You can not use any built-in Python functions except for `len`.
2. Write a function that takes a possibly empty string as a parameter and recursively produces a reversed copy of the string.
3. Similar to the last problem, write a function that takes a possibly empty *list* as a parameter (the list may contain *any* Python data types), and recursively produces a reversed copy of the list.
4. Read the following function and see if you can figure out what it does, then run it.

```
import turtle

def draw(length, n):
    if n == 0:
        return
    angle = 50
    turtle.forward(length*n)
    turtle.left(angle)
    draw(length, n-1)
    turtle.right(2*angle)
    draw(length, n-1)
    turtle.left(angle)
    turtle.backward(length*n)

draw(10, 4)
turtle.done()
```

5. The Koch curve is a fractal that looks something like this:

   To draw a Koch curve with length x, all you have to do is

   (a) Draw a Koch curve with length $x/3$.

   (b) Turn left 60 degrees.

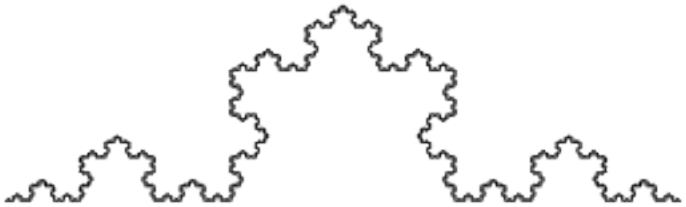


Figure 9.2: Koch curve fractal.

(c) Draw a Koch curve with length $x/3$.

(d) Turn right 120 degrees.

(e) Draw a Koch curve with length $x/3$.

(f) Turn left 60 degrees.

(g) Draw a Koch curve with length $x/3$.

The only exception is if x is less than 3. In that case, you can just draw a straight line with length x.

(a) Write a function called `koch` that takes a turtle and a length as parameters, and that uses the turtle to draw a Koch curve with the given length.

(b) Write a function called `snowflake` that draws three Koch curves to make the outline of a snowflake.

(c) The Koch curve can be generalized in several ways. See http://wikipedia.org/wiki/Koch_snowflake for examples and implement your favorite.

6. The Ackermann function, $A(m, n)$, is defined [3] as:

   $A(m, n) =$

   - $n + 1$ if $m = 0$
   - $A(m - 1, 1)$ if $m > 0 and n = 0$
   - $A(m - 1, A(m, n - 1))$ if $m > 0$ and $n > 0$

```
Write a function named 'ack' that evaluates Ackerman's function.
Use your function to evaluate 'ack(3, 4)', which should be 125. What
happens for larger values of 'm' and 'n'?
```

7. A number, a, is a power of b if it is divisible by b and a/b is a power of b. Write a function called `is_power` that takes parameters `a` and `b` and returns `True` if `a` is a power of `b`.

8. The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder [4].

   One way to find the GCD of two numbers is Euclid's algorithm, which is based on the observation that if r is the remainder when a is divided by b, then $gcd(a, b) = gcd(b, r)$. As a base case, we can consider $gcd(a, 0) = a$.

   Write a function called `gcd` that takes parameters `a` and `b` and returns their greatest common divisor. If you need help, see http://wikipedia.org/wiki/Euclidean_algorithm.

[3] See http://wikipedia.org/wiki/Ackermann_function.

[4] This exercise is based on an example from Abelson and Sussman's *Structure and Interpretation of Computer Programs*.

CHAPTER 10

File input and output

10.1 Reading word lists

Several of the examples and exercises in this chapter require a (big) list of English words. There are many of word lists available on the Internet, but the one most suitable for our purpose is one of the word lists collected and contributed to the public domain by Grady Ward as part of the Moby lexicon project [1]. It includes is a list of 117,969 official crosswords; that is, words that are considered valid in crossword puzzles and other word games. I've extracted and posted this list at: http://cs.colgate.edu/~jsommers/cosc101/words.txt

This file is in *plain text* (as the suffix `.txt` suggests), so you can open it with a text editor, but you can also read it from Python. The built-in function `open` takes the name of the file as a parameter and returns a **file object** you can use to read the file.

```
>>> fin = open('words.txt')
>>> print fin
<open file 'words.txt', mode 'r' at 0xb7f4b380>
```

`fin` is a common name for a file object used for input ("*f*ile *in*put"). Mode `'r'` indicates that this file is open for reading (as opposed to `'w'` for writing).

The file object provides several methods for reading, including `readline`, which reads characters from the file until it gets to a newline and returns the result as a string:

```
>>> fin.readline()
'aa\r\n'
```

The first word in this particular list is "aa," which is a kind of lava. The sequence `\r\n` represents two whitespace characters, a carriage return (`\r`) and a newline (`\n`), that separate this word from the next.

The file object keeps track of where it is in the file, so if you call `readline` again, you get the next word:

```
>>> fin.readline()
'aah\r\n'
```

The next word is "aah," which is a perfectly legitimate word, so stop looking at me like that. If it's the whitespace that's bothering you, we can get rid of it with the string method `strip`:

[1] http://wikipedia.org/wiki/Moby_Project.

```
>>> line = fin.readline()
>>> word = line.strip()
>>> print word
aahed
```

You can also use a file object as part of a `for` loop. This program reads `words.txt` and prints each word, one per line:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print word
fin.close()
```

The above program also includes a call to the `close` method as the last statement. You should always remember to close files when you're done with them!

Example:

1. Write a program that reads `words.txt` and prints only the words with more than 20 characters (not counting whitespace). At the end of your program, you should also print out the number of words that are longer than 20 characters (not including whitespace).

10.2 Practice exercises

There are solutions to these exercises in the next section. You should at least attempt each one before you read the solutions.

1. In 1939 Ernest Vincent Wright published a 50,000 word novel called *Gadsby* that does not contain the letter "e." Since 'e' is the most common letter in English, that's not easy to do.

   In fact, it is difficult to construct a solitary thought without using that most common symbol. It is slow going at first, but with caution and hours of training you can gradually gain facility.

   All right, I'll stop now.

   Write a function called `has_no_e` that returns `True` if the given word doesn't have the letter 'e' in it.

2. Write a program that uses your `has_no_e` function to print only the words in `words.txt` that have no `e`. You should also compute and print the percentage of words in the file that have no 'e'.

3. Write a function named `avoids` that takes a word and a string of forbidden letters, and that returns `True` if the word doesn't use any of the forbidden letters.

4. Write a program to prompt the user to enter a string of forbidden letters and print the number of words in `words.txt` that do not contain any of them.

   Fun challenge: can you find a combination of 5 forbidden letters that excludes the *smallest* number of words?

5. Write a function named `uses_only` that takes a word and a string of letters, and that returns `True` if the word contains only letters in the list.

   Fun challenge: can you make a sentence using only the letters `acefhlo`? Other than "Hoe alfalfa?"

6. Write a function named `uses_all` that takes a word and a string of required letters, and that returns `True` if the word uses all the required letters at least once. How many words are there that use all the vowels `aeiou`? How about `aeiouy`?

7. Write a function called `is_abecedarian` that returns `True` if the letters in a word appear in alphabetical order (double letters are ok). How many abecedarian words are there?

10.3 Search

All of the exercises in the previous section have something in common; they can be solved with the search pattern we previously saw with the `find` function we wrote in the strings chapter. The simplest example is:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

The `for` loop traverses the characters in `word`. If we find the letter "e", we can immediately return `False`; otherwise we have to go to the next letter. If we exit the loop normally, that means we didn't find an "e", so we return `True`.

`avoids` is a more general version of `has_no_e` but it has the same structure:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

We can return `False` as soon as we find a forbidden letter; if we get to the end of the loop, we return `True`.

`uses_only` is similar except that the sense of the condition is reversed:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Instead of a list of forbidden letters, we have a list of available letters. If we find a letter in `word` that is not in `available`, we can return `False`.

`uses_all` is similar except that we reverse the role of the word and the string of letters:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

Instead of traversing the letters in `word`, the loop traverses the required letters. If any of the required letters do not appear in the word, we can return `False`.

If you were really thinking like a computer scientist, you would have recognized that `uses_all` was an instance of a previously-solved problem, and you would have written:

```
def uses_all(word, required):
    return uses_only(required, word)
```

This is an example of a program development method called **problem recognition**, which means that you recognize the problem you are working on as an instance of a previously-solved problem, and apply a previously-developed solution.

10.4 Looping with indices

I wrote the functions in the previous section with `for` loops because I only needed the characters in the strings; I didn't have to do anything with the indices.

For `is_abecedarian` we have to compare adjacent letters, which is a little tricky with a `for` loop:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

An alternative is to use recursion:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Another option is to use a `while` loop:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

The loop starts at `i=0` and ends when `i=len(word)-1`. Each time through the loop, it compares the ith character (which you can think of as the current character) to the $i + 1$th character (which you can think of as the next).

If the next character is less than (alphabetically before) the current one, then we have discovered a break in the abecedarian trend, and we return `False`.

If we get to the end of the loop without finding a fault, then the word passes the test. To convince yourself that the loop ends correctly, consider an example like `'flossy'`. The length of the word is 6, so the last time the loop runs is when `i` is 4, which is the index of the second-to-last character. On the last iteration, it compares the second-to-last character to the last, which is what we want.

Here is a version of `is_palindrome` that uses two indices; one starts at the beginning and goes up; the other starts at the end and goes down.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

Or, if you noticed that this is an instance of a previously-solved problem, you might have written:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

10.5 Reading and writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file earlier.

To write a file, you have to open it with mode `'w'` as a second parameter:

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

The `write` method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

Again, the file object keeps track of where it is, so if you call `write` again, it adds the new data to the end.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
```

As we saw with reading, when you are done writing, you should close the file.

```
>>> fout.close()
```

10.6 The `format` method for strings

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with the `str` conversion function:

```
>>> x = 52
>>> f.write(str(x))
```

An alternative is to use the **"format" method** on strings. The string on which the format method is called should contain **replacement fields** surrounded by curly braces (`{}`). Arguments to the `format` method are inserted in the replacement fields, in order.

Here are some examples:

```
"My name is {}!".format('Tim!')
```

which results in the string `'My name is Tim!'`

```
'''{} is the answer to life,
   the universe,
   and something else, maybe'''.format(41)
```

which results in `'41 is the answer to life, the universe, and something else, maybe'`

Within the curly braces, you can specify *how* the replacement item should be formatted. For example, you can specify that replacement items should be centered, left justified, or right justified within some column width, or that a floating point number be shown with a certain number of decimal places:

```
    'I am {:d} years old in dog years'.format(age * 7)

Assuming ``age`` is defined, this will convert ``age * 7`` to a
decimal integer (that's the ``d`` in the replacement field). If
``age`` is 2, the resulting string is just
```'I am 14 years old in dog years'```

    'Center this: {:^30}'.format('my string')

In this example, the caret character (```^```) means to center the
replacement item, and the value 30 is the field width. So the
string ``my string`` is centered in a 30-character width. In
addition to ```^```, you can use ``<`` to left-justify an item,
and ``>`` to right-justify an item.

    'PI to 3 decimal places is {:.3f}'.format(math.pi)

In this example, we specify that we want to convert the
replacement item to a floating point number (the ``f``), and
show 3 decimal places (the ``.3`` preceding the ``f``).

    coords = [4.2, 5.532]
    'x,y = {:.1f},{:.1f}'.format(coords[0], coords[1])

In this example, we have two replacement fields, each with
floating point format specifiers. Because we have two
replacement fields, we need two replacement items as arguments
to the ``format`` method.
```

The `format` method is useful, but the replacement field syntax is a bit complex, and we won't go into any more depth here. For full details, please refer to the Python documentation: http://docs.python.org/library/string.html#formatstrings. (Finally, note that if you're using a version of Python less than 2.7, the format method works a bit differently. Please ensure that you're using Python 2.7.)

## 10.7 Filenames and paths

Files are organized into **directories** (also called "folders"). Every running program has a "current directory," which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The `os` module provides functions for working with files and directories ("os" stands for "operating system"). `os.getcwd` returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/Users/jsommers
```

`cwd` stands for "current working directory." The result in this example is `/Users/jsommers`, which is the home directory of a user named `jsommers`.
```

A string like `cwd` that identifies a file is called a **path**. A **relative path** starts from the current directory; an **absolute path** starts from the topmost directory in the file system.

The paths we have seen so far are simple filenames, so they are relative to the current directory. To find the absolute path to a file, you can use `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/Users/jsommers/memo.txt'
```

`os.path.exists` checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, `os.path.isdir` checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

Similarly, `os.path.isfile` checks whether it's a file.

`os.listdir` returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dir):
    for name in os.listdir(dir):
        path = os.path.join(dir, name)

        if os.path.isfile(path):
            print path
        else:
            walk(path)
```

`os.path.join` takes a directory and a file name and joins them into a complete path.

Exercise

1. Modify `walk` so that instead of printing the names of the files, it returns a list of names.

   The `os` module provides a function called `walk` that is similar to this one but more versatile. Read the documentation and use it to print the names of the files in a given directory and its subdirectories.

10.8 Catching exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an `IOError`:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>> fout = open('/etc/passwd', 'w')
IOError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/Users')
IOError: [Errno 21] Is a directory
```

To avoid these errors, you could use functions like `os.path.exists` and `os.path.isfile`, but it would take a lot of time and code to check all the possibilities (if "`Errno 21`" is any indication, there are at least 21 things that can go wrong).

It is better to go ahead and try, and deal with problems if they happen, which is exactly what the `try` statement does. The syntax is similar to an `if` statement:

```
try:
    fin = open('bad_file')
    for line in fin:
        print line
    fin.close()
except:
    print 'Something went wrong.'
```

Python starts by executing the `try` clause. If all goes well, it skips the `except` clause and proceeds. If an exception occurs, it jumps out of the `try` clause and executes the `except` clause.

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

10.9 Case study 1: retrieving and processing files available on the internet

The `urllib2` module provides methods for manipulating URLs and downloading files from the internet. Interestingly, opening a URL on the internet using this module is very similar to opening a file stored on your own computer.

At the beginning of this chapter, we saw that a copy of the `words.txt` file is stored at http://cs.colgate.edu/~jsommers/cosc101/words.txt. Using the `urllib2` module, we can open and process the file, even though it isn't locally stored!

```
import urllib2

connection = urllib2.urlopen("http://cs.colgate.edu/~jsommers/cosc101/words.txt")
for line in connection:
    print line
connection.close()
```

The only difference between this program and an equivalent program that reads a locally stored file is how the file is opened! (And, of course, we need to import the `urllib2` module). Note that if you run the above program, you'll see each word in the `words.txt` file, followed by a blank line. Make sure you understand *why* that's the case, and that you know how to modify the above program to *only* print each word on a line (and not the blank lines).

10.10 Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named `wc.py` with the following code:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print linecount('wc.py')
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc
7
```

Now you have a module object `wc`:

```
>>> print wc
<module 'wc' from 'wc.py'>
```

That provides a function called `linecount`:

```
>>> wc.linecount('wc.py')
7
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it executes the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't execute them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    print linecount('wc.py')
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value `__main__`; in that case, the test code is executed. Otherwise, if the module is being imported, the test code is skipped.

Example:

1. Type this example into a file named `wc.py` and run it as a script. Then run the Python interpreter and `import wc`. What is the value of `__name__` when the module is being imported?

Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function `reload`, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

10.11 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print s
1 2  3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print repr(s)
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies might cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at http://wikipedia.org/wiki/Newline. Or, of course, you could write one yourself.

10.12 Glossary

file object: A value that represents an open file.

format string: A string, used with the format method, that contains replacement fields.

replacement field: A sequence of characters in a format string, like `{:d}`, or `{:.2f}`, or even just `{}`, that specifies how a replacement item should be formatted.

text file: A sequence of characters stored in permanent storage like a hard drive.

directory: A named collection of files, also called a folder.

path: A string that identifies a file.

relative path: A path that starts from the current directory.

absolute path: A path that starts from the topmost directory in the file system.

catch: To prevent an exception from terminating a program using the `try` and `except` statements.

10.13 Exercises

1. This question is based on a Puzzler that was broadcast on the radio program *Car Talk* [2]:

   Give me a word with three consecutive double letters. I'll give you a couple of words that almost qualify, but don't. For example, the word committee, c-o-m-m-i-t-t-e-e. It would be great except for the 'i' that sneaks in there. Or Mississippi: M-i-s-s-i-s-s-i-p-p-i. If you could take out those i's it would work. But there is a word that has three consecutive pairs of letters and to the best of my knowledge this may be the only word. Of course there are probably 500 more but I can only think of one. What is the word?

   Write a program to find it.

2. Here's another *Car Talk* Puzzler [3]:

[2] http://www.cartalk.com/content/puzzler/transcripts/200725
[3] http://www.cartalk.com/content/puzzler/transcripts/200803

"I was driving on the highway the other day and I happened to notice my odometer. Like most odometers, it shows six digits, in whole miles only. So, if my car had 300,000 miles, for example, I'd see 3-0-0-0-0-0.

"Now, what I saw that day was very interesting. I noticed that the last 4 digits were palindromic; that is, they read the same forward as backward. For example, 5-4-4-5 is a palindrome, so my odometer could have read 3-1-5-4-4-5.

"One mile later, the last 5 numbers were palindromic. For example, it could have read 3-6-5-4-5-6. One mile after that, the middle 4 out of 6 numbers were palindromic. And you ready for this? One mile later, all 6 were palindromic!

"The question is, what was on the odometer when I first looked?"

Write a Python program that tests all the six-digit numbers and prints any numbers that satisfy these requirements.

3. Here's another *Car Talk* Puzzler you can solve with a search [4]:

   "Recently I had a visit with my mom and we realized that the two digits that make up my age when reversed resulted in her age. For example, if she's 73, I'm 37. We wondered how often this has happened over the years but we got sidetracked with other topics and we never came up with an answer.

   "When I got home I figured out that the digits of our ages have been reversible six times so far. I also figured out that if we're lucky it would happen again in a few years, and if we're really lucky it would happen one more time after that. In other words, it would have happened 8 times over all. So the question is, how old am I now?"

   Write a Python program that searches for solutions to this Puzzler. Hint: you might find the string method `zfill` useful.

4. The website http://www.uszip.com provides information about every zip code in the country. For example, the URL http://www.uszip.com/zip/13346 provides information about Hamilton, NY, including population, longitude and latitude, etc.

   Using the `urllib2` module, write a program that prompts the user for a zip code and prints the name and population of the corresponding town.

   Note: the text you get from uszip.com is in HTML, the language most web pages are written in. Even if you don't know HTML, you should be able to extract the information you are looking for.

   By the way, your program is an example of a "screen scraper." You can read more about this term at http://wikipedia.org/wiki/Screen_scraping.

5. In a large collection of MP3 files, there may be more than one copy of the same song, stored in different directories or with different file names. The goal of this exercise is to search for these duplicates.

   (a) Write a program that searches a directory and all of its subdirectories, recursively, and returns a list of complete paths for all files with a given suffix (like `.mp3`). Hint: `os.path` provides several useful functions for manipulating file and path names.

   (b) To recognize duplicates, you can use a hash function that reads the file and generates a short summary of the contents. For example, MD5 (Message-Digest algorithm 5) takes an arbitrarily-long "message" and returns a 128-bit "checksum." The probability is very small that two files with different contents will return the same checksum. You can read about MD5 at http://wikipedia.org/wiki/Md5.

   To obtain the MD5 checksum on the contents of a file, you can use the `hashlib` module, built in to Python:

```
>>> import hashlib
>>> csum = hashlib.md5()
>>> csum.update("Nobody inspects the spammish repetition.")
```

[4] http://www.cartalk.com/content/puzzler/transcripts/200813

```
>>> csum.hexdigest()
'dc6480df97e6f16ec0aa18c96522aee6'
```

With the `update` method on the `csum` object, you can update the checksum by adding new strings (or file contents). When you're done processing the contents of a file, you can use the `hexdigest` method to obtain the final checksum in hexadecimal form.

CHAPTER 11

Lists

In this chapter, we go into more depth on lists. Lists are an incredibly useful data structure for solving a variety of problems. Before we dive in, you may wish to review the earlier section on lists, which appeared at the beginning of the "iteration" chapter.

11.1 List slices

The slice operator we've used for strings also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists. [1]

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

[1] The admonition "Do not fold, spindle, or mutilate" was printed on the punch cards that were used by early computers. See http://www.alteich.com/tidbits/t042202.htm and http://design.osu.edu/carlson/history/PDFs/lubar-hollerith.pdf

11.2 List methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

`extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

This example leaves `t2` unmodified.

`sort` arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

Note that the append, extend, and sort methods do not return anything (except None). So, a statement like:

```
t = t.sort()
```

will result in the list `t` reassigned to `None`.

There is a `sorted` *function* built in to Python that takes a list as a parameter, and returns a new, sorted list. The original list is unchanged:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> sorted(t)
['a', 'b', 'c', 'd', 'e']
>>> print t
['d', 'c', 'e', 'b', 'a']
```

There is also a `reverse` method for lists, and a `reversed` *function* that work somewhat similarly to `sort` and `sorted`. The `reversed` function, instead of returning a list, returns an *iterator*. The way to turn the iterator into a list is to compose the list function with the `reversed` function:

```
>>> t = ['a', 'b', 'c', 'd', 'e']
>>> t.reverse()
>>> print t
['e', 'd', 'c', 'b', 'a']
>>> reversed(t)
<listreverseiterator object at 0x10f1e56d0>
>>> list(reversed(t))
['a', 'b', 'c', 'd', 'e']
>>> print t
['e', 'd', 'c', 'b', 'a']
```

The `count` method, similar to strings, takes one item as a parameter and returns an integer count of occurrences of all identical items in the list:

```
>>> t = ['d','c','e','b','a','c','d','c']
>>> t.count('d')
2
>>> t.count('a')
1
>>> t.count('x')
0
```

The `index` method, again, similar to strings, takes up to two parameters. The first parameter is an item to search for, and the second (optional) parameter is the starting index to search from. If a second parameter is not specified, the default value of 0 is used. Importantly, if an item you search for (i.e., the first parameter to `index`) is *not* in the list, you'll get a `ValueError` exception. Note that there is no `find` method for lists.

```
>>> t.index('c')
1
>>> t.index('c',3)
5
>>> t.index('x')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'x' is not in list
```

11.3 Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
```

`pop` modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

The return value from `remove` is `None`.

To remove more than one element, you can use `del` with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
```

```
>>> print t
['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

11.4 Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid `l` because it looks too much like `1`. So that's why many of the examples in this chapter use `t`.

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method, as we saw in the strings chapter:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
```

An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` is the inverse of `split`. It takes a list of strings and concatenates the elements. `join` is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In this case the delimiter is a space character, so `join` puts a space between words. To concatenate strings without spaces, you can use the empty string, `''`, as a delimiter.

11.5 Objects and values

If we execute these assignment statements:

```
a = 'banana'
b = 'banana'
```

We know that `a` and `b` both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states:

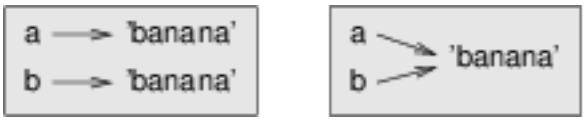


Figure 11.1: Variables referring to different objects, and variables that refer to the same object.

In one case, `a` and `b` refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the `is` operator.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

In this example, Python only created one string object, and both `a` and `b` refer to it.

But when you create two lists, you get two objects:

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

So the state diagram looks like this:

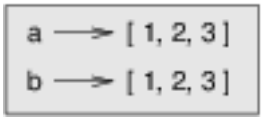


Figure 11.2: Variables that refer to two different list objects.

In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value. If you execute `[1,2,3]`, you get a list object whose value is a sequence of integers. If another list has the same elements, we say it has the same value, but it is not the same object.

11.6 Aliasing

If `a` refers to an object and you assign `b = a`, then both variables refer to the same object:

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

The state diagram looks like this:

The association of a variable with an object is called a **reference**. In this example, there are two references to the same object.

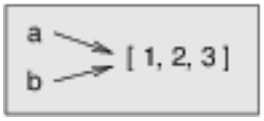


Figure 11.3: Variables that are "aliases" of each other; they refer to the same list object.

An object with more than one reference has more than one name, so we say that the object is **aliased**.

If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana'
b = 'banana'
```

It almost never makes a difference whether `a` and `b` refer to the same string or not.

11.7 List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

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

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

The parameter `t` and the variable `letters` are aliases for the same object. The stack diagram looks like this:

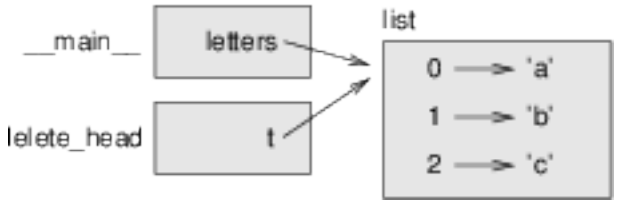


Figure 11.4: Variable `letters` and parameter variable `t` refer to the same list object in memory.

Since the list is shared by two frames, I drew it between them.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None

>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3]
>>> t2 is t3
False
```

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]               # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

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

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

Examples:

1. Write a function called `chop` that takes a list and modifies it, removing the first and last elements, and returns `None`.
2. Write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements.

11.8 Searching for items in a list

Say that we have a very large list of words called `wordlist`, and we want to check whether the word `zyzzy` is in the list. We have a few options so far:

1. The `in` operator.
2. The list `count` method (if the count is greater than zero, the word is in the list).
3. The list `index` method.
4. A "homegrown" function that searches the list, one word after another.

The first two of the four options above just tell us whether the word is in the list, and the last two can tell us the *location* (index) of the word in the list. Taking approach 3, if we wanted to write a function called `findWord` that returns the index of the word in the list (or -1 if the word isn't found), here's how we could do it:

```
def findWord(wordlist, word):
    '''
    Check whether word is in the wordlist.  Return
    the index of the word if found, or -1 otherwise.
    '''
    try:
        # The index method causes a ValueError exception
        # if the item we're searching for is not found.
        return wordlist.index(word)
    except:
        # if we get here, we know the word isn't in
        #  the list, so just return -1.
        return -1
```

Examples:

1. Write a `findWord2` function to implement option 4: it shouldn't use any list methods while doing the same thing as `findWord`. That is, it should search the wordlist for the word and return the index where it occurs. If the word isn't found, the function should return -1.
2. Say we have a list of 100,000 words. If the word we're looking for is not in the list, how many comparisons between the word we're searching for and a word in the list do we need to make? (This is the *worst case* situation for trying to find a word. It should be pretty easy to identify the number of comparisons we have to make.)

For all four options above, the worst case is pretty ugly: we have to search the entire list. This search technique is called **linear search**, because we have to linearly search each and every item in the list. Even with a fast, modern computer, linearly searching a very large list is best avoided if possible.

What if we were to simply *sort* the list of words? If the list is in alphabetical order, we can speed things up with a **bisection search** (also known as **binary search**), which is similar to what you do when you look a word up in the dictionary. You start in the middle and check to see whether the word you are looking for comes before the word in the middle of the list. If so, then you search the first half of the list the same way. Otherwise you search the second half.

Either way, you cut the remaining search space in half. If the word list has 115,000 words, it will take about 17 steps to find the word or conclude that it's not there. This is a huge improvement over the worst case with linear search! Writing a function to perform a binary search is left as an exercise, and interestingly, binary search lends itself nicely to a recursive implementation.

You should also know that Python includes a `bisect` module that contains an implementation of binary search. You can check out the documentation for it here: http://docs.python.org/library/bisect.html.

11.9 Map, filter and reduce

To add up all the numbers in a list, you can use a loop like this:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

`total` is initialized to 0. Each time through the loop, `x` gets one element from the list.

As the loop executes, `total` accumulates the sum of the elements; a variable used this way is sometimes called an **accumulator**.

Adding up the elements of a list is such a common operation that Python provides the built-in function, `sum`:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

An operation like this that combines a sequence of elements into a single value is sometimes called **reduce**.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` is initialized with an empty list; each time through the loop, we append the next element. So `res` is another kind of accumulator.

An operation like `capitalize_all` is sometimes called a **map** because it "maps" a function (in this case the method `capitalize`) onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` is a string method that returns `True` if the string contains only upper case letters.

An operation like `only_upper` is called a **filter** because it selects some of the elements and filters out the others.

11.9.1 First class functions

This is an advanced topic, but fits well with the idea of *mapping* and lists.

Let's say we wanted to generalize the mapping function we wrote above, to apply different transformations to lists of strings. Our original example was to produce a new list in which each original string is capitalized, but we might want to apply different functions, such as changing all the strings to lower case, to upper case, or to obtain a list of all the string lengths. What we'd like to avoid is to write separate functions for each of these tasks.

In Python, functions are objects that can be passed as parameters to other functions. To generalize the `capitalize_all` function, what we can do is pass in a mapping function as a second parameter to a more general function we'll call `mapper`. To each item in the input list, we'll apply our mapping function, which will produce a new item to be added onto a result list:

```
def mapper(input_list, mapping_function):
    result = []
    for value in input_list:
        new_value = mapping_function(value)
        result.append(new_value)
    return result

def tolower(s):
    return s.lower()
```

```
t = ['apple','banana','kiwi','star fruit']
lower_list = mapper(t, tolower)
```

So above, `tolower` is our mapping function that gets passed into the `mapper`. Inside `mapper`, each item in the original list is transformed by the mapping function and added on to a new list.

This is great — we've generalized the idea of mapping. But we can do even better.

It seems a waste to write a two-line function like `tolower`, when it is so simple. To simplify this part of our program, we can use what is called a **lambda** in Python. Lambdas are short, "anonymous" functions that can be defined at the point at which they're needed. The syntax for writing a lambda function is to use the `lambda` keyword, followed by a comma-separated list of parameters, a colon (`:`), and a short (at most one line) Python expression. Importantly, there is no explicit return statement in a `lambda`. Python just uses the result of the expression as the return value.

For example, here is a function defined with `lambda` that takes two parameters, and returns their sum. We assign the result of the `lambda` expression to the variable `addtwo`. The result is just a function object, just like any function object that gets created with `def`:

```
>>> addtwo = lambda x, y: x + y
>>> addtwo
<function <lambda> at 0x10ca0e938>
>>> addtwo(3,5)
8
```

Now, to modify our example above using a `lambda` expression (and adding on a couple more examples of mapping):

```
def mapper(input_list, mapping_function):
    result = []
    for value in input_list:
        new_value = mapping_function(value)
        result.append(new_value)
    return result

t = ['apple','banana','kiwi','star fruit']
upper_list = mapper(t, lambda s: s.upper())
lower_list = mapper(t, lambda s: s.lower())
len_list = mapper(t, lambda s: len(s))
```

As you can see, using `lambda` makes our mapping code more compact, and still fairly easy to read and understand.

11.10 Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Don't forget that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

   If you are used to writing string code like this:

   ```
   word = word.strip()
   ```

   It is tempting to write list code like this:

   ```
   t = t.sort()            # WRONG!
   ```

   Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at http://docs.python.org/lib/typesseq.html. The methods and operators that only apply to mutable sequences are documented at http://docs.python.org/lib/typesseq-mutable.html.

2. Pick an idiom and stick with it.

   Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment.

   To add an element, you can use the `append` method or the + operator. Assuming that `t` is a list and `x` is a list element, these are right:

   ```
   t.append(x)
   t = t + [x]
   ```

   And these are wrong:

   ```
   t.append([x])          # WRONG!
   t = t.append(x)        # WRONG!
   t + [x]                # WRONG!
   t = t + x              # WRONG!
   ```

   Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

   If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

   ```
   orig = t[:]
   t.sort()
   ```

   In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using `sorted` as a variable name!

11.11 Glossary

list: A sequence of values.

element: One of the values in a list (or other sequence), also called items.

index: An integer value that indicates an element in a list.

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

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

mapping: A relationship in which each element of one set corresponds to an element of another set. For example, a list is a mapping from indices to elements.

accumulator: A variable used in a loop to add up or accumulate a result.

augmented assignment: A statement that updates the value of a variable using an operator like +=.

reduce: A processing pattern that traverses a sequence and accumulates the elements into a single result.

map: A processing pattern that traverses a sequence and performs an operation on each element.

filter: A processing pattern that traverses a list and selects the elements that satisfy some criterion.

object: Something a variable can refer to. An object has a type and a value.

equivalent: Having the same value.

identical: Being the same object (which implies equivalence).

reference: The association between a variable and its value.

aliasing: A circumstance where two or more variables refer to the same object.

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

linear search: The approach of searching for an item in a list by inspecting each element in the list, one after another.

binary search: Also known as *bisection search*. An approach for searching a list which assumes the items in the list in sorted order. It proceeds by checking the middle element, and deciding whether the item to search for is in the first half or second half of the list. The "search space" is repeatedly cut in half by applying this same idea to smaller and smaller portions of the list.

11.12 Exercises

1. Write a function called `is_sorted` that takes a list as a parameter and returns `True` if the list is sorted in ascending order and `False` otherwise. You can assume (as a precondition) that the elements of the list can be compared with the relational operators <, >, etc.

   For example, `is_sorted([1,2,2])` should return `True` and `is_sorted(['b','a'])` should return `False`.

2. Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called `is_anagram` that takes two strings and returns `True` if they are anagrams.

3. The (so-called) Birthday Paradox:

   Write a function called `has_duplicates` that takes a list and returns `True` if there is any element that appears more than once. It should not modify the original list.

   If there are 23 students in your class, what are the chances that two of you have the same birthday? You can estimate this probability by generating random samples of 23 birthdays and checking for matches. Hint: you can generate random birthdays with the `randint` function in the `random` module.

   You can read about this problem at http://wikipedia.org/wiki/Birthday_paradox.

4. Write a function called `remove_duplicates` that takes a list and returns a new list with only the unique elements from the original. Hint: they don't have to be in the same order.

5. Write a function that reads the file `words.txt` and builds a list with one element per word. Write two versions of this function, one using the `append` method and the other using the idiom `t = t + [x]`. Which one takes longer to run? Why?

6. Two words are a "reverse pair" if each is the reverse of the other. Write a program that finds all the reverse pairs in the word list.

7. Two words "interlock" if taking alternating letters from each forms a new word [2]. For example, "shoe" and "cold" interlock to form "schooled."

   (a) Write a program that finds all pairs of words that interlock. Hint: don't enumerate all pairs!

   (b) Can you find any words that are three-way interlocked; that is, every third letter forms a word, starting from the first, second or third?

8. Binary search implementation.

[2] This exercise is inspired by an example at http://puzzlers.org.

(a) Write a function called `bisect_iterative` that takes a sorted list and a target value and *iteratively* finds and returns the index of the value in the list, if it's there, or `None` if it's not. You should use a `while` loop and no recursion in this version.

(b) Write a function called `bisect_recursive` that takes a sorted list and a target value and *recursively* finds and returns the index of the value in the list, if it's there, or `None` if it's not. There should not be any explicit loops in your solution.

CHAPTER 12

Dictionaries

A **dictionary** is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called **keys**) and a set of values. Each key maps to a value. The association of a key and a value is called a **key-value pair** or sometimes an **item**. Because of the way that dictionaries work, they're also called associative arrays.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items. Because `dict` is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print eng2sp
{}
```

The squiggly-brackets, `{}`, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key `'one'` to the value `'uno'`. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print eng2sp
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

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

But if you print `eng2sp`, you might be surprised:

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

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

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

The key `'two'` always maps to the value `'dos'` so the order of the items doesn't matter.

If the key isn't in the dictionary, you get an exception:

```
>>> print eng2sp['four']
KeyError: 'four'
```

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

```
>>> len(eng2sp)
3
```

The `in` operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method `values`, which returns the values as a list, and then use the `in` operator:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

The `in` operator uses different algorithms for lists and dictionaries. For lists, it uses a search algorithm, as in the `find` function we wrote earlier. As the list gets longer, the search time gets longer in direct proportion. For dictionaries, Python uses an algorithm called a **hashtable** that has a remarkable property: the `in` operator takes about the same amount of time no matter how many items there are in a dictionary. I won't explain how that's possible, but you can read more about it at http://wikipedia.org/wiki/Hash_table.

Example:

1. Write a function that reads the words in `words.txt` and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the `in` operator as a fast way to check whether a string is in the dictionary.

12.1 Dictionary as a set of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function `ord`), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

The name of the function is **histogram**, which is a statistical term for a set of counters (or frequencies).

The first line of the function creates an empty dictionary. The `for` loop traverses the string. Each time through the loop, if the character `c` is not in the dictionary, we create a new item with key `c` and the initial value 1 (since we have seen this letter once). If `c` is already in the dictionary we increment `d[c]`.

Here's how it works:

```
>>> h = histogram('brontosaurus')
>>> print h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters `'a'` and `'b'` appear once; `'o'` appears twice, and so on.

Example:

1. Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

   ```
   >>> h = histogram('a')
   >>> print h
   {'a': 1}
   >>> h.get('a', 0)
   1
   >>> h.get('b', 0)
   0
   ```

   Use `get` to write `histogram` more concisely. You should be able to eliminate the `if` statement.

12.2 Looping and dictionaries

If you use a dictionary in a `for` statement, it traverses the keys of the dictionary. For example, `print_hist` prints each key and the corresponding value:

```
def print_hist(h):
    for c in h:
        print c, h[c]
```

Here's what the output looks like:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
```

```
r 2
t 1
o 1
```

Again, the keys are in no particular order.

Example:

1. Dictionaries have a method called `keys` that returns the keys of the dictionary, in no particular order, as a list.

   Modify `print_hist` to print the keys and their values in alphabetical order.

12.3 Reverse lookup

Given a dictionary `d` and a key `k`, it is easy to find the corresponding value `v = d[k]`. This operation is called a **lookup**.

But what if you have `v` and you want to find `k`? You have two problems: first, there might be more than one key that maps to the value `v`. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup**; you have to search.

Here is a function that takes a value and returns the first key that maps to that value:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise ValueError
```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before, `raise`. The `raise` statement causes an exception; in this case it causes a `ValueError`, which generally indicates that there is something wrong with the value of a parameter.

If we get to the end of the loop, that means `v` doesn't appear in the dictionary as a value, so we raise an exception.

Here is an example of a successful reverse lookup:

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> print k
r
```

And an unsuccessful one:

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in reverse_lookup
ValueError
```

The result when you raise an exception is the same as when Python raises one: it prints a traceback and an error message.

The `raise` statement takes a detailed error message as an optional argument. For example:

```
>>> raise ValueError, 'value does not appear in the dictionary'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: value does not appear in the dictionary
```

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

Example:

1. Modify `reverse_lookup` so that it builds and returns a list of *all* keys that map to `v`, or an empty list if there are none.

12.4 Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you were given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters.

Here is a function that inverts a dictionary:

```
def invert_dict(d):
    inv = dict()
    for key in d:
        val = d[key]
        if val not in inv:
            inv[val] = [key]
        else:
            inv[val].append(key)
    return inv
```

Each time through the loop, `key` gets a key from `d` and `val` gets the corresponding value. If `val` is not in `inv`, that means we haven't seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```
>>> hist = histogram('parrot')
>>> print hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inv = invert_dict(hist)
>>> print inv
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

And here is a diagram showing `hist` and `inv`:

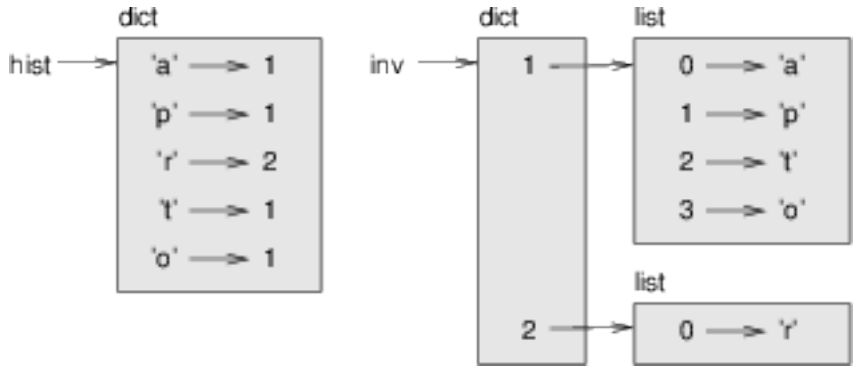


Figure 12.1: The dictionary `hist` and its "inverted" equivalent `inv`.

A dictionary is represented as a box with the type `dict` above it and the key-value pairs inside. If the values are

integers, floats or strings, I usually draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple.

Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be **hashable**.

A **hash** is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly.

That's why the keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use tuples, which we will see in the next chapter.

Since dictionaries are mutable, they can't be used as keys, but they *can* be used as values.

12.5 Memos

If you played with the `fibonacci function <#sec:fibonacci>`_, 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.

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

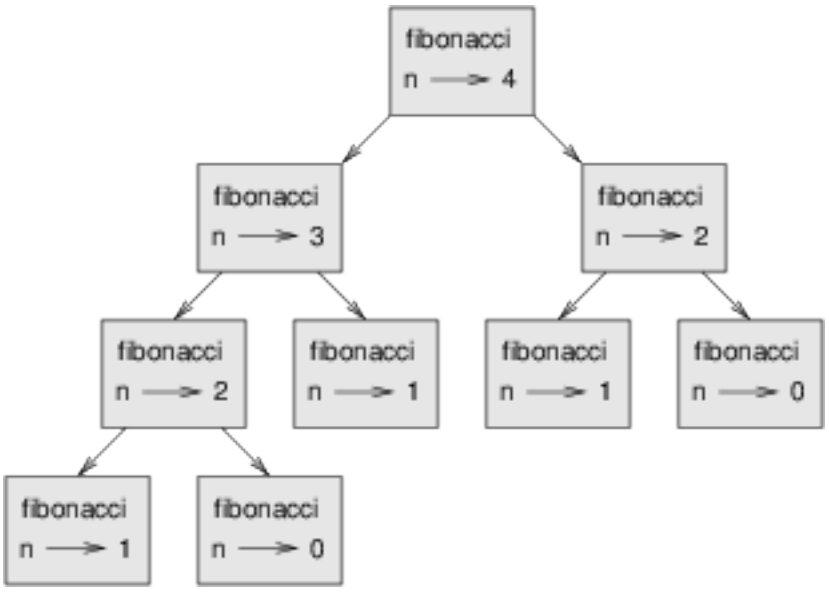


Figure 12.2: Fibonacci function call graph.

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

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

One 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 **memo** [1]. Here is an implementation of `fibonacci` using memos:

```
known = {0:0, 1:1}

def fibonacci(n):
    if n in known:
        return known[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

`known` is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

Whenever `fibonacci` is called, it checks `known`. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it.

Example:

1. Run this version of `fibonacci` and the original with a range of parameters and compare their run times.

12.6 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

- Scale down the input:
  - If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first `n` lines.
  - If there is an error, you can reduce `n` to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.
- Check summaries and types:
  - Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.
  - A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.
- Write self-checks:
  - Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a "sanity check" because it detects results that are "insane."

[1] See http://wikipedia.org/wiki/Memoization.

  - Another kind of check compares the results of two different computations to see if they are consistent. This is called a "consistency check."

Again, time you spend building scaffolding can reduce the time you spend debugging.

12.7 Glossary

dictionary: A mapping from a set of keys to their corresponding values.

key-value pair: The representation of the mapping from a key to a value.

item: Another name for a key-value pair.

key: An object that appears in a dictionary as the first part of a key-value pair.

value: An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word "value."

implementation: A way of performing a computation.

hashtable: The algorithm used to implement Python dictionaries.

hash function: A function used by a hashtable to compute the location for a key.

hashable: A type that has a hash function. Immutable types like integers, floats and strings are hashable; mutable types like lists and dictionaries are not.

lookup: A dictionary operation that takes a key and finds the corresponding value.

reverse lookup: A dictionary operation that takes a value and finds one or more keys that map to it.

singleton: A list (or other sequence) with a single element.

call graph: A diagram that shows every frame created during the execution of a program, with an arrow from each caller to each callee.

histogram: A set of counters.

memo: A computed value stored to avoid unnecessary future computation.

12.8 Exercises

1. Use a dictionary to write a faster, simpler version of `has_duplicates`. This function should take a list as a parameter and return `True` if there is any object that appears more than once in the list.

2. Two words are "rotate pairs" if you can rotate one of them and get the other.

   Write a program that reads a wordlist and finds all the rotate pairs.

3. Here's another Puzzler from *Car Talk* [2]:

   "This was sent in by a fellow named Dan O'Leary. He came upon a common one-syllable, five-letter word recently that has the following unique property. When you remove the first letter, the remaining letters form a homophone of the original word, that is a word that sounds exactly the same. Replace the first letter, that is, put it back and remove the second letter and the result is yet another homophone of the original word. And the question is, what's the word?

   "Now I'm going to give you an example that doesn't work. Let's look at the five-letter word, 'wrack.' W-R-A-C-K, you know like to 'wrack with pain.' If I remove the first letter, I am left with a four-letter word, 'R-A-C-K.'

[2] http://www.cartalk.com/content/puzzler/transcripts/200717.

As in, 'Holy cow, did you see the rack on that buck! It must have been a nine-pointer!' It's a perfect homophone. If you put the 'w' back, and remove the 'r,' instead, you're left with the word, 'wack,' which is a real word, it's just not a homophone of the other two words.

"But there is, however, at least one word that Dan and we know of, which will yield two homophones if you remove either of the first two letters to make two, new four-letter words. The question is, what's the word?"

To check whether two words are homophones, you can use the CMU Pronouncing Dictionary. You can download it from http://www.speech.cs.cmu.edu/cgi-bin/cmudict. (The file you're looking for is a text file named `cmudict.0.7a`.)

Write a program that lists all the words that solve the Puzzler.

CHAPTER 13

Tuples

13.1 Tuples are immutable

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that *tuples are immutable*.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include a final comma:

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

A value in parentheses is not a tuple:

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

Another way to create a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print t
()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

Because `tuple` is the name of a built-in function, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator is used to index an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

And the slice operator selects a range of elements.

```
>>> print t[1:3]
('b', 'c')
```

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

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

13.2 Tuple assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap `a` and `b`:

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

This solution is cumbersome; **tuple assignment** is more elegant:

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

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

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

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
>>> print uname
monty
>>> print domain
python.org
```

13.3 Tuples as return values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute `x/y` and then `x%y`. It is better to compute them both at the same time.

The built-in function `divmod` takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

```
>>> t = divmod(7, 3)
>>> print t
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
>>> print quot
2
>>> print rem
1
```

Here is an example of a function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```

`max` and `min` are built-in functions that find the largest and smallest elements of a sequence. `min_max` computes both and returns a tuple of two values.

13.4 Variable-length argument tuples

Functions can take a variable number of arguments. A parameter name that begins with `*` **gathers** arguments into a tuple. For example, `printall` takes any number of arguments and prints them:

```
def printall(*args):
    print args
```

The gather parameter can have any name you like, but `args` is conventional. Here's how the function works:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

The complement of gather is **scatter**. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the `*` operator. For example, `divmod` takes exactly two arguments; it doesn't work with a tuple:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

```
>>> divmod(*t)
(2, 1)
```

```
**Example**:
```

```
1. Many of the built-in functions use variable-length argument
   tuples. For example, ``max`` and ``min`` can take any number of
   arguments:

   ::

       >>> max(1,2,3)
       3

   But ``sum`` does not.

   ::

       >>> sum(1,2,3)
       TypeError: sum expected at most 2 arguments, got 3

   Write a function called ``sumall`` that takes any number of
   arguments and returns their sum.
```

13.5 Lists and tuples

`zip` is a built-in function that takes two or more sequences and "zips" them into a list [1] of tuples where each tuple contains one element from each sequence.

This example zips a string and a list:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples where each tuple contains a character from the string and the corresponding element from the list.

If the sequences are not the same length, the result has the length of the shorter one.

```
>>> zip('Anne', 'Elk')
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a `for` loop to traverse a list of tuples:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print number, letter
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to `letter` and `number`. The output of this loop is:

```
0 a
1 b
2 c
```

If you combine `zip`, `for` and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, `has_match` takes two sequences, `t1` and `t2`, and returns `True` if there is an index `i` such that `t1[i] == t2[i]`:

[1] In Python 3.0, `zip` returns an iterator of tuples, but for most purposes, an iterator behaves like a list.

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function `enumerate`:

```
for index, element in enumerate('abc'):
    print index, element
```

The output of this loop is:

```
0 a
1 b
2 c
```

Again.

13.6 Dictionaries and tuples

Dictionaries have a method called `items` that returns a list of tuples, where each tuple is a key-value pair [2].

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> print t
[('a', 0), ('c', 2), ('b', 1)]
```

As you should expect from a dictionary, the items are in no particular order.

Conversely, you can use a list of tuples to initialize a new dictionary:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

Combining `dict` with `zip` yields a concise way to create a dictionary:

```
>>> d = dict(zip('abc', range(3)))
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method `update` also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

Combining `items`, tuple assignment and `for`, you get the idiom for traversing the keys and values of a dictionary:

```
for key, val in d.items():
    print val, key
```

The output of this loop is:

```
0 a
2 c
1 b
```

[2] This behavior is slightly different in Python 3.0.

Again.

It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined `last`, `first` and `number`, we could write:

```
directory[(last,first)] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

```
for last, first in directory:
    print first, last, directory[(last,first)]
```

This loop traverses the keys in `directory`, which are tuples. It assigns the elements of each tuple to `last` and `first`, then prints the name and corresponding telephone number.

There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple `('Cleese', 'John')` would appear:

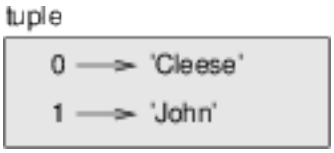


Figure 13.1: 2-tuple example

But in a larger diagram you might want to leave out the details. For example, a diagram of the telephone directory might appear:

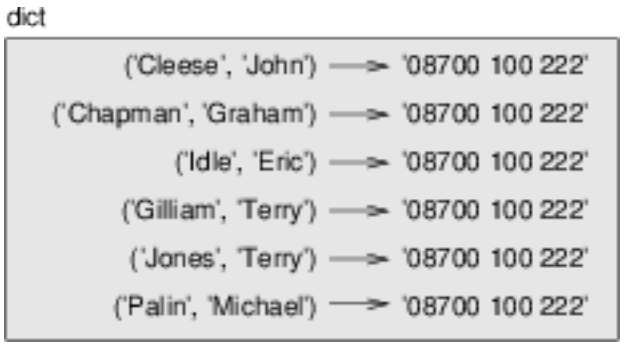


Figure 13.2: Another tuple example

Here the tuples are shown using Python syntax as a graphical shorthand.

The telephone number in the diagram is the complaints line for the BBC, so please don't call it.

13.7 Comparing tuples

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

The `sort` function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.

This feature lends itself to a pattern called **DSU** for

Decorate a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

Sort the list of tuples, and

Undecorate by extracting the sorted elements of the sequence.

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
def sort_by_length(words):
    t = []
    for word in words:
        t.append((len(word), word))

    t.sort(reverse=True)

    res = []
    for length, word in t:
        res.append(word)
    return res
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length.

`sort` compares the first element, length, first, and only considers the second element to break ties. The keyword argument `reverse=True` tells `sort` to go in decreasing order.

The second loop traverses the list of tuples and builds a list of words in descending order of length.

> **Example**:
>
> In this example, ties are broken by comparing words, so words with the same length appear in reverse alphabetical order. For other applications you might want to break ties at random. Modify this example so that words with the same length appear in random order. Hint: see the `random` function in the `random` module.

13.8 Sequences of sequences

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how and why do you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like `sort` and `reverse`, which modify existing lists. But Python provides the built-in functions `sorted` and `reversed`, which take any sequence as a parameter and return a new list with the same elements in a different order.

13.9 Debugging

Lists, dictionaries and tuples are known generically as **data structures**; in this chapter we are starting to see compound data structures, like lists of tuples, and dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call **shape errors**; that is, errors caused when a data structure has the wrong type, size or composition. For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

A strategy for ferreting out such problems is to use the `assert` function discussed earlier to test the shape of data structures that are passed in to functions. Python includes a function called `isinstance` that can test whether a variable is an instance of a particular data type. You can compose that with `assert` to ensure that the data types your function receives are what you expect.

```
>>> x = []
>>> isinstance(x, list)
True
>>> y = {}
>>> assert(isinstance(y, list))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

If you create assertions like this and they fail, you know that one of the parameters to a function is of the wrong shape. You can then search for instances where the function is called to find and fix situations in which the function is called incorrectly.

13.10 Glossary

tuple: An immutable sequence of elements.

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

gather: The operation of assembling a variable-length argument tuple.

scatter: The operation of treating a sequence as a list of arguments.

DSU: Abbreviation of "decorate-sort-undecorate," a pattern that involves building a list of tuples, sorting, and extracting part of the result.

data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.

shape (of a data structure): A summary of the type, size and composition of a data structure.

13.11 Exercises

1. Write a function called `most_frequent` that takes a string and prints the letters in decreasing order of frequency. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at http://wikipedia.org/wiki/Letter_frequencies.

2. Write a program that reads a word list from a file (see this section) and prints all the sets of words that are anagrams.

   Here is an example of what the output might look like:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   Hint: you might want to build a dictionary that maps from a set of letters to a list of words that can be spelled with those letters. The question is, how can you represent the set of letters in a way that can be used as a key?

3. Modify the previous program so that it prints the largest set of anagrams first, followed by the second largest set, and so on.

4. In Scrabble, a "bingo" is when you play all seven tiles in your rack along with a letter on the board to form an eight-letter word. What set of 8 letters forms the most possible bingos? Hint: there are seven.

5. Two words form a "metathesis pair" if you can transform one into the other by swapping two letters [3]; for example, "converse" and "conserve." Write a program that finds all of the metathesis pairs in the dictionary. Hint: don't test all pairs of words, and don't test all possible swaps.

6. Here's another Car Talk Puzzler [4]:

   "What is the longest English word, that remains a valid English word, as you remove its letters one at a time?

   "Now, letters can be removed from either end, or the middle, but you can't rearrange any of the letters. Every time you drop a letter, you wind up with another English word. If you do that, you're eventually going to wind up with one letter and that too is going to be an English word—one that's found in the dictionary. I want to know what's the longest word and how many letters does it have?

   I'm going to give you a little modest example: Sprite. Ok? You start off with sprite, you take a letter off, one from the interior of the word, take the r away, and we're left with the word spite, then we take the e off the end, we're left with spit, we take the s off, we're left with pit, it, and I."

   Write a program to find all words that can be reduced in this way, and then find the longest one.

   This exercise is a little more challenging than most, so here are some suggestions:

   - You might want to write a function that takes a word and computes a list of all the words that can be formed by removing one letter. These are the "children" of the word.
   - Recursively, a word is reducible if any of its children are reducible. As a base case, you can consider the empty string reducible.
   - The wordlist I provided, `words.txt`, doesn't contain single letter words. So you might want to add "I", "a", and the empty string.
   - To improve the performance of your program, you might want to memoize the words that are known to be reducible.

[3] This exercise is inspired by an example at http://puzzlers.org.
[4] http://www.cartalk.com/content/puzzler/transcripts/200651

CHAPTER 14

Case study: data structure selection

14.1 Word frequency analysis

As usual, you should at least attempt the following exercises before you read the sections that follow.

Examples:

1. Write a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.

   Hint: The `string` module provides strings named `whitespace`, which contains space, tab, newline, etc., and `punctuation` which contains the punctuation characters. Let's see if we can make Python swear:

   ```
   >>> import string
   >>> print string.punctuation
   !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
   ```

   Also, you might consider using the string methods `strip`, `replace` and `translate`.

2. Go to Project Gutenberg (http://gutenberg.org) and download your favorite out-of-copyright book in plain text format.

   Modify your program from the previous exercise to read the book you downloaded, skip over the header information at the beginning of the file, and process the rest of the words as before.

3. Then modify the program to count the total number of words in the book, and the number of times each word is used.

4. Print the number of different words used in the book. Compare different books by different authors, written in different eras. Which author uses the most extensive vocabulary?

5. Modify the program from the previous exercise to print the 20 most frequently-used words in the book.

6. Modify the previous program to read a word list and then print all the words in the book that are not in the word list. How many of them are typos? How many of them are common words that *should* be in the word list, and how many of them are really obscure?

7. Write a function named `choose_from_hist` that takes a histogram as defined in the dictionary chapter and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:

```
>>> t = ['a', 'a', 'b']
>>> h = histogram(t)
>>> print h
{'a': 2, 'b': 1}
```

your function should `'a'` with probability $2/3$ and `'b'` with probability $1/3$.

You will probably find the `random.choice` function in the `random` module helpful for this exercise. This function takes a list as a parameter, and returns one of the items from the list, chosen uniformly at random.

14.2 Word histogram

Here is a program that reads a file and builds a histogram of the words in the file:

```
import string

def process_file(filename):
    h = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, h)
    return h

def process_line(line, h):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()

        h[word] = h.get(word, 0) + 1

hist = process_file('emma.txt')
```

This program reads `emma.txt`, which contains the text of * Emma* by Jane Austen.

`process_file` loops through the lines of the file, passing them one at a time to `process_line`. The histogram `h` is being used as an accumulator.

`process_line` uses the string method `replace` to replace hyphens with spaces before using `split` to break the line into a list of strings. It traverses the list of words and uses `strip` and `lower` to remove punctuation and convert to lower case. (It is a shorthand to say that strings are "converted;" remember that string are immutable, so methods like `strip` and `lower` return new strings.)

Finally, `process_line` updates the histogram by creating a new item or incrementing an existing one.

To count the total number of words in the file, we can add up the frequencies in the histogram:

```
def total_words(h):
    return sum(h.values())
```

The number of different words is just the number of items in the dictionary:

```
def different_words(h):
    return len(h)
```

Here is some code to print the results:

```
print 'Total number of words:', total_words(hist)
print 'Number of different words:', different_words(hist)
```

And the results:

```
Total number of words: 161073
Number of different words: 7212
```

14.3 Most common words

To find the most common words, we can apply the DSU pattern; `most_common` takes a histogram and returns a list of word-frequency tuples, sorted in reverse order by frequency:

```
def most_common(h):
    t = []
    for key, value in h.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

Here is a loop that prints the ten most common words:

```
t = most_common(hist)
print 'The most common words are:'
for freq, word in t[0:10]:
    print word, '\t', freq
```

And here are the results from *Emma*:

```
The most common words are:
to      5242
the     5204
and     4897
of      4293
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

14.4 Optional parameters

We have seen built-in functions and methods that take a variable number of arguments. It is possible to write user-defined functions with optional arguments, too. For example, here is a function that prints the most common words in a histogram

```
def print_most_common(hist, num=10)
    t = most_common(hist)
    print 'The most common words are:'
    for freq, word in t[0:num]:
        print word, '\t', freq
```

The first parameter is required; the second is optional. The **default value** of `num` is 10.

If you only provide one argument:

```
print_most_common(hist)
```

`num` gets the default value. If you provide two arguments:

```
print_most_common(hist, 20)
```

`num` gets the value of the argument instead. In other words, the optional argument **overrides** the default value.

If a function has both required and optional parameters, all the required parameters have to come first, followed by the optional ones.

14.5 Dictionary subtraction

Finding the words from the book that are not in the word list from `words.txt` is a problem you might recognize as set subtraction; that is, we want to find all the words from one set (the words in the book) that are not in another set (the words in the list).

`subtract` takes dictionaries `d1` and `d2` and returns a new dictionary that contains all the keys from `d1` that are not in `d2`. Since we don't really care about the values, we set them all to None.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

To find the words in the book that are not in `words.txt`, we can use `process_file` to build a histogram for `words.txt`, and then subtract:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print "The words in the book that aren't in the word list are:"
for word in diff.keys():
    print word,
```

Here are some of the results from *Emma*:

```
The words in the book that aren't in the word list are:
 rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Some of these words are names and possessives. Others, like "rencontre," are no longer in common use. But a few are common words that should really be in the list!

Example:

1. Python provides a data structure called `set` that provides many common set operations. Read the documentation at http://docs.python.org/lib/types-set.html and write a program that uses set subtraction to find words in the book that are not in the word list.

14.6 Random words

To choose a random word from the histogram, the simplest algorithm is to build a list with multiple copies of each word, according to the observed frequency, and then choose from the list:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

The expression `[word] * freq` creates a list with `freq` copies of the string `word`. The `extend` method is similar to `append` except that the argument is a sequence.

Example:

1. This algorithm works, but it is not very efficient; each time you choose a random word, it rebuilds the list, which is as big as the original book. An obvious improvement is to build the list once and then make multiple selections, but the list is still big.

   An alternative is:

   (a) Use `keys` to get a list of the words in the book.

   (b) Build a list that contains the cumulative sum of the word frequencies. The last item in this list is the total number of words in the book, n.

   (c) Choose a random number from 1 to n. Use a bisection search to find the index where the random number would be inserted in the cumulative sum.

   (d) Use the index to find the corresponding word in the word list.

   Write a program that uses this algorithm to choose a random word from the book.

14.7 Markov analysis

If you choose words from the book at random, you can get a sense of the vocabulary, you probably won't get a sentence:

> this the small regard harriet which knightley's it most things

A series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you would expect an article like "the" to be followed by an adjective or a noun, and probably not a verb or adverb.

One way to measure these kinds of relationships is Markov analysis [1], which characterizes, for a given sequence of words, the probability of the word that comes next. For example, the song *Eric, the Half a Bee* begins:

> Half a bee, philosophically, Must, ipso facto, half not be. But half the bee has got to be Vis a vis, its entity. D'you see? But can a bee be said to be Or not to be an entire bee When half the bee is not a bee Due to some ancient injury?

In this text, the phrase "half the" is always followed by the word "bee," but the phrase "the bee" might be followed by either "has" or "is".

The result of Markov analysis is a mapping from each prefix (like "half the" and "the bee") to all possible suffixes (like "has" and "is").

[1] This case study is based on an example from Kernighan and Pike, *The Practice of Programming*, 1999.

Given this mapping, you can generate a random text by starting with any prefix and choosing at random from the possible suffixes. Next, you can combine the end of the prefix and the new suffix to form the next prefix, and repeat.

For example, if you start with the prefix "Half a," then the next word has to be "bee," because the prefix only appears once in the text. The next prefix is "a bee," so the next suffix might be "philosophically," "be" or "due."

In this example the length of the prefix is always two, but you can do Markov analysis with any prefix length. The length of the prefix is called the "order" of the analysis.

Markov analysis:

1. Write a program to read a text from a file and perform Markov analysis. The result should be a dictionary that maps from prefixes to a collection of possible suffixes. The collection might be a list, tuple, or dictionary; it is up to you to make an appropriate choice. You can test your program with prefix length two, but you should write the program in a way that makes it easy to try other lengths.
2. Add a function to the previous program to generate random text based on the Markov analysis. Here is an example from *Emma* with prefix length 2:

   > He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.

   For this example, I left the punctuation attached to the words. The result is almost syntactically correct, but not quite. Semantically, it almost makes sense, but not quite.

   What happens if you increase the prefix length? Does the random text make more sense?
3. Once your program is working, you might want to try a mash-up: if you analyze text from two or more books, the random text you generate will blend the vocabulary and phrases from the sources in interesting ways.

14.8 Data structures

Using Markov analysis to generate random text is fun, but there is also a point to this exercise: data structure selection. In your solution to the previous exercises, you had to choose:

- How to represent the prefixes.
- How to represent the collection of possible suffixes.
- How to represent the mapping from each prefix to the collection of possible suffixes.

Ok, the last one is the easy; the only mapping type we have seen is a dictionary, so it is the natural choice.

For the prefixes, the most obvious options are string, list of strings, or tuple of strings. For the suffixes, one option is a list; another is a histogram (dictionary).

How should you choose? The first step is to think about the operations you will need to implement for each data structure. For the prefixes, we need to be able to remove words from the beginning and add to the end. For example, if the current prefix is "Half a," and the next word is "bee," you need to be able to form the next prefix, "a bee."

Your first choice might be a list, since it is easy to add and remove elements, but we also need to be able to use the prefixes as keys in a dictionary, so that rules out lists. With tuples, you can't append or remove, but you can use the addition operator to form a new tuple:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` takes a tuple of words, `prefix`, and a string, `word`, and forms a new tuple that has all the words in `prefix` except the first, and `word` added to the end.

For the collection of suffixes, the operations we need to perform include adding a new suffix (or increasing the frequency of an existing one), and choosing a random suffix.

Adding a new suffix is equally easy for the list implementation or the histogram. Choosing a random element from a list is easy; choosing from a histogram is harder to do efficiently.

So far we have been talking mostly about ease of implementation, but there are other factors to consider in choosing data structures. One is run time. Sometimes there is a theoretical reason to expect one data structure to be faster than other; for example, I mentioned that the `in` operator is faster for dictionaries than for lists, at least when the number of elements is large.

But often you don't know ahead of time which implementation will be faster. One option is to implement both of them and see which is better. This approach is called **benchmarking**. A practical alternative is to choose the data structure that is easiest to implement, and then see if it is fast enough for the intended application. If so, there is no need to go on. If not, there are tools, like the `profile` module, that can identify the places in a program that take the most time.

The other factor to consider is storage space. For example, using a histogram for the collection of suffixes might take less space because you only have to store each word once, no matter how many times it appears in the text. In some cases, saving space can also make your program run faster, and in the extreme, your program might not run at all if you run out of memory. But for many applications, space is a secondary consideration after run time.

One final thought: in this discussion, I have implied that we should use one data structure for both analysis and generation. But since these are separate phases, it would also be possible to use one structure for analysis and then convert to another structure for generation. This would be a net win if the time saved during generation exceeded the time spent in conversion.

14.9 Debugging

When you are debugging a program, and especially if you are working on a hard bug, there are four things to try:

- *Reading*:
  - Examine your code, read it back to yourself, and check that it says what you meant to say.
- *Running*:
  - Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.
- *Ruminating*:
  - Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?
- *Retreating*:
  - At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can starting rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming," which is the process of making

random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

Taking a break helps with the thinking. So does talking. If you explain the problem to someone else (or even yourself), you will sometimes find the answer before you finish asking the question.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can paste the pieces back in a little bit at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

14.10 Glossary

default value: The value given to an optional parameter if no argument is provided.

override: To replace a default value with an argument.

benchmarking: The process of choosing between data structures by implementing alternatives and testing them on a sample of the possible inputs.

14.11 Exercises

1. The "rank" of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.

   Zipf's law describes a relationship between the ranks and frequencies of words in natural languages [2]. Specifically, it predicts that the frequency, f, of the word with rank r is:

   $$f = cr^{-s}$$

   where s and c are parameters that depend on the language and the text. If you take the logarithm of both sides of this equation, you get:

   $$\log f = \log c - s \log r$$

   So if you plot $\log f$ versus $\log r$, you should get a straight line with slope $-s$ and intercept $\log c$.

   Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency, with $\log f$ and $\log r$. Use the graphing program of your choice to plot the results and check whether they form a straight line. Can you estimate the value of s?

[2] See http://wikipedia.org/wiki/Zipf's_law.

CHAPTER 15

Classes and objects

We have used many of Python's built-in types, like integers, strings, lists, and dictionaries. Each instance of a type in Python (e.g., a specific string, integer, or list) is called an **object**. Although the term "object" sounds generic, it has a very specific meaning in software design. In fact, what we'll be learning about in this chapter is a style of programming and program design called **object-oriented programming**, or OOP. Before we get into designing an object-oriented program, we will first learn about how new types — and new types of objects — can be defined and created.

15.1 User-defined types

As an initial example, we'll create a new type called `Point` that represents a point in two-dimensional space. 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.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, `x` and `y`.
- We could store the coordinates as elements in a list or tuple.
- We could create a dictionary that has two keys, `x` and `y`, with corresponding values.
- **We could create a *new type* to represent points as objects.** (Woo-hoo! Pick me! Pick me!)

Creating a new type is (a little bit) more complicated than the other options, but it has advantages that will be apparent soon.

Every type in Python is defined by the type's **class**. You can think of a class as a blueprint or model from which objects can be created. A minimal class definition looks like this:

```
class Point(object):
    """represents a point in 2-D space"""
```

This header indicates that the new class is a `Point`, which is a kind of `object`, which is a built-in type.

The body is a docstring that explains the purpose of the class. Normally, you will also define functions and variables inside a class definition; we will get to that shortly.

Defining a class named `Point` creates a class object.

```
>>> print Point
<class '__main__.Point'>
```

Because `Point` is defined at the top level, its "full name" is `__main__.Point`.

Besides being like a blueprint, the class object is also like a factory for creating objects. To create a Point, you call `Point` as if it were a function.

```
>>> blank = Point()
>>> print blank
<__main__.Point instance at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to `blank`. Creating a new object is called **instantiation**, and the object is an **instance** of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the string starting `0x` is a memory location in hexadecimal format—base 16).

Even though it might feel a little strange using the class name as a function, you can construct any built-in Python type using the same syntax:

```
>>> mylist = list()
>>> print list
[]
```

Using the class/type name as a function works for *any* type in Python: `int`, `str`, `list`, `dict`, `tuple`, etc. And although printing the `list` object looks a little prettier than our `Point` object, we'll make the `Point` look better soon.

15.2 Attributes and Methods

We've just made a `Point` class from which we can construct `Point` instances, or `Point` objects. Objects, in the peculiar programming language sense, can have **attributes** and **methods** associated with them.

- You can think of the **attributes** as data, or information, stored inside an object. In object-oriented programming languages, attributes are also referred to as **instance variables**.
- We've already used **methods** on objects. These are functions that are, in a sense, *attached* to an object. In Python, a method is invoked by using the dot notation syntax:

  ```
  >>> # object.method(parameters)
  >>> s = "aabbcc"
  >>> s.count('b')
  2
  ```

In the above example with the string object `s`, we invoke the `count` method on the object, passing the string `b` as a parameter. You can think of the attributes for the string object as being the sequence of characters that make up the string.

15.2.1 Adding attributes

You can assign new attributes to an instance using dot notation and the assignment operator:

```
>>> blank = Point()
>>> blank.x = 3.0
>>> blank.y = 4.0
```

The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**:

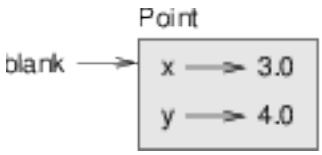


Figure 15.1: Class diagram of a `point` object.

The variable `blank` refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

You can read the value of an attribute 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 variable named `x`. There is no conflict between the variable `x` and the attribute `x`.

Interestingly (and usefully), *objects are mutable* — we can change the values of attributes:

```
>>> blank.x = 5.5
>>> blank.y = blank.x * 2
```

15.2.2 Adding methods

Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit. As with functions, we use the `def` keyword to define methods, but the method `def` header needs to be indented inside the class definition.
- The syntax for invoking a method is different from the syntax for calling a function.

Let's get started writing a method to set the `x` and `y` attributes in the object to new values:

```
class Point(object):
    '''represents a point in 2-D space'''

    def setXY(point, x, y):
        '''Set values for x and y attributes.
           Parameters:
              point is the object we're invoking this method on.
              x is the new value for the x attribute.
              y is the new value for the y attribute.
           There's no return value.'''
        point.x = x
        point.y = y
```

We might use our `Point` class to create an object and set its `x` and `y` attributes using the `setXY` method as follows:

```
>>> p = Point()
>>> p.setXY(8.0, 7.5)
>>> print p.x
8.0
```

```
>>> print p.y
7.5
```

On line 2 of the above code, `setXY` is the name of the method, and `p` is the object on which the method is invoked, which is also called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the `setXY` method, the subject is assigned to the first parameter, so in this case `p` is assigned to `point`.

By convention in Python, the first parameter of a method is called `self`, so the Pythonically correct way to write `setXY` would be:

```
class Point(object):
    '''represents a point in 2-D space'''

    def setXY(self, x, y):
        '''Set values for x and y attributes.'''
        self.x = x
        self.y = y
```

15.2.3 The init method

Instead of making a `setXY` method to initialize the attributes of our `Point` class, a more conventional way to set initial attribute values is to create a special method called the **constructor**, **c'tor**, or **initializer**. The method name for the constructor is *always* `__init__` in Python, and it is automatically invoked when an object is instantiated. Constructors are used for initializing attributes in an object, and to perform any other initialization that might be required when a new instance is created.

Let's modify our `Point` class to include an `__init__` method that accepts two parameters for initializing our `x` and `y` coordinates. We'll still retain the `setXY` method, too.

```
class Point(object):
    '''represents a point in 2-D space'''

    def __init__(self, x, y):
        '''Point constructor: takes initial x,y values'''
        self.x = x
        self.y = y

    def setXY(self, x, y):
        self.x = x
        self.y = y
```

To create a new `Point` object, we have to change our call to `Point` to pass in initial values for `x` and `y`:

```
>>> p = Point(3.2, 8.9)
>>> print p.x
3.2
>>> print p.y
8.9
```

Since `__init__` and `setXY` are nearly identical, we could even refine our code a bit to reduce redundancy:

```
class Point(object):
    '''represents a point in 2-D space'''

    def __init__(self, x, y):
        '''Point constructor: sets initial x,y values'''
```

```
        self.setXY(x, y)

    def setXY(self, x, y):
        self.x = x
        self.y = y
```

The optimization isn't particularly large in this example, but it is still a good idea to avoid repeating the same code. Also, if we add any new attributes, we only have to specify their initialization in *one* place.

15.2.4 Additional `Point` methods

Let's add to our `Point` class by writing two more methods:

- A `getXY` method that doesn't take any parameters and returns a tuple consisting of the `x` and `y` coordinates, and
- a `distance` method that takes another `Point` object as a parameter and computes and returns the Euclidean distance between the *subject* `Point` (the `Point` object on which the `distance` method is called) and the `Point` object passed as the parameter.

First, the `getXY` method:

```
class Point(object):

    # ... other methods defined in Point

    def getXY(self):
        ''' return the x,y coordinates
            as a tuple.'''
        return (self.x, self.y)
```

Although we said about that the `getXY` method doesn't take any parameters, *all* methods must *always* take at least one parameter: the subject, or `self` object. Inside the method, we simply return a tuple consisting of the `x` and `y` components.

In a program, we might use the `getXY` method as follows:

```
>>> p = Point(5,2)
>>> coord_tuple = p.getXY()
>>> print coord_tuple
(5,2)
```

Now, for the `distance` method:

```
import math

class Point(object):

    # ... other methods defined in Point

    def distance(self, other):
        ''' compute and return the Euclidean
            distance between this point and another.'''
        d = (self.x - other.x)**2 + (self.y - other.y)**2
        return math.sqrt(d)
```

In a program, we might use the `distance` method as follows:

```
>>> p1 = Point(5,1)
>>> p2 = Point(3,7)
>>> d = p1.distance(p2)
>>> print d
6.324555320336759
```

15.2.5 Printing objects

__str__ is a special method, like __init__, that is supposed to return a string representation of an object. For the Point class, we might write the __str__ method as follows:

```
class Point(object):

    # ... other methods defined in Point

    def __str__(self):
        return "Point ({:.1f},{:.1f})".format(self.x, self.y)
```

When you print an object, Python automatically and implicitly invokes the __str__ method:

```
>>> print p1
'Point (5.0,1.0)'
>>> print p2
Point (3.0,7.0)
```

When you write a new class, a good idea is to start by writing __init__, which makes it easier to instantiate objects, and __str__, which is useful for debugging.

Note that any method names that are prefixed and suffixed with __ are called **magic methods** in Python. They're "magic" because they're invoked automatically and implicitly by Python: a programmer generally never explicitly invokes these methods.

15.2.6 The full Point class

Putting all our work together, here is the full definition of the Point class that we created:

```
import math

class Point(object):
    '''represents a point in 2-D space'''

    def __init__(self, x, y):
        '''Point constructor: takes initial x,y values'''
        self.x = x
        self.y = y

    def setXY(self, x, y):
        '''Set x and y coordinates to new values.'''
        self.x = x
        self.y = y

    def getXY(self):
        ''' return the x,y coordinates as a tuple.'''
        return (self.x, self.y)

    def distance(self, other):
```

```
        ''' compute and return the Euclidean
            distance between this point and another.'''
        d = (self.x - other.x)**2 + (self.y - other.y)**2
        return math.sqrt(d)

    def __str__(self):
        return "Point ({:.1f},{:.1f})".format(self.x, self.y)
```

15.3 Object-oriented program design

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, and most of the computation is expressed in terms of operations on objects.
- Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

For example, the `Point` class defined above corresponds to the mathematical concept of a point.

For solving problems in an object-oriented programming style, the main idea is to model the entities or concepts in the problem domain, including *attributes* that are stored by the entity, and *actions*, or *methods* that can be performed by the entities. Our goal in this course is for you to get your feet wet with OOP ideas; later courses go into more depth on OOP design.

15.4 A `Rectangle` class

Let's try to test our knowledge so far by designing a class that models a rectangle.

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For the rectangle class we're designing, what attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.

Here is the class definition, starting with `__init__` and `__str__`:

```
class Rectangle(object):
    """represent a rectangle.
       attributes: width, height, corner.
    """

    def __init__(self, width, height, corner):
        self.width = width
        self.height = height
        self.corner = corner

    def __str__(self):
```

```
        return "Rectangle lower-left: ({:.1f},{:.1f}) "
               "upper-right: ({:.1f},{:.1f})".format(self.corner.x,
               self.corner.y, self.corner.x + self.width,
               self.corner.y+self.height)
```

Once we create the `Rectangle` class, we might use the two methods we've written to construct and print a rectangle object:

```
>>> r = Rectangle(100.0, 200.0, Point(0, 0))
>>> print r
Rectangle lower-left: (0.0, 0.0) upper-right: (100.0, 200.0)
```

The figure shows the state of this object:

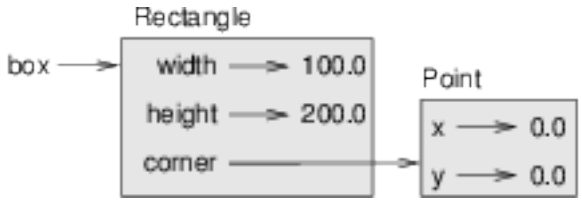


Figure 15.2: Diagram of a `rectangle` object that refers to a `point` object.

An object that is an attribute of another object is **embedded**: the `Point` object that represents the lower-left corner of the rectangle is *embedded* in the `Rectangle` object. This sort of relationship is also referred to as a **HAS-A** relationship in object-oriented programming. In this case, a `Rectangle` HAS-A `Point`.

Examples:

1. Write a method named `move_rectangle` that takes two numbers named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the `x` coordinate of `corner` and adding `dy` to the `y` coordinate of `corner`.
2. Write a method named `perimeter` that computes and returns the perimeter length of the rectangle.
3. Write a method named `area` that computes and returns the area of the rectangle.

15.5 A `Circle` class

Since we've started on shapes, how about making a class to model a circle. Our choices for attributes are little simpler than with the rectangle. It probably makes sense to have a `Point` attribute that represents the center of the circle, and a number that holds either the radius or diameter of the circle. Before you look carefully at the code below, see if you can write out the class definition for `Circle`, including the constructor and `__str__` magic method.

```
class Circle(object):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def __str__(self):
        return "Circle ({:.1f},{:.1f}) with radius {:.1f}".format(
            self.center.x, self.center.y, self.radius)

**Examples**:

1. Write a method named ``move_circle`` that takes two numbers named
```

```
   ``dx`` and ``dy``. It should change the center position of the
   center by adding ``dx`` to the ``x`` coordinate of ``center`` and
   adding ``dy`` to the ``y`` coordinate of ``center``.

2. Write a method named ``perimeter`` that computes and returns the
   circumference of the circle.

3. Write a method named ``area`` that computes and returns the area
   of the circle.
```

15.6 Inheritance

If you've faithfully done the examples above (do them now if you haven't already!), you may have noticed some similarities in how they are implemented. For one, the `move_...` methods are remarkably similar. Also, even though the `perimeter` and `area` methods for the `Rectangle` and `Circle` are *implemented* differently, they have the same name, and (at least in an abstract way) are doing the same things. This should not be surprising, since circles and rectangles are both shapes.

Besides HAS-A relationships in object-oriented programming, there are also **IS-A** relationships that are often directly supported through programming language features. In our `Shape` example, a circle IS-A shape, and a rectangle IS-A shape. In object-oriented programming languages, IS-A relationships are directly supported through a featured called **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class. It is called "inheritance" because the new class inherits the methods of the existing class. Extending this metaphor, the existing class is called the **parent** and the new class is called the **child**.

In the examples below, we'll design a parent `Shape` class, and refactor (revise) our `Rectangle` and `Circle` classes so that they inherit from `Shape`.

15.6.1 A `Shape` class

Let's first make our amorphous shape class. Just to make things somewhat interesting, let's give shapes a name and color. We'll also define `area` and `perimeter` methods; they can just return 0.

```
def Shape(object):
    '''A generic shape class.'''
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def __str__(self):
        return "I am a {} {}.".format(self.color, self.name)

    def area(self):
        return 0.0

    def perimeter(self):
        return 0.0
```

15.6.2 Refactoring `Rectangle`

Now, let's modify the `Rectangle` class so that it inherit from `Shape`. We'll start with the `__init__` method:

```
class Rectangle(Shape):
    def __init__(self, corner, width, height, color):
        Shape.__init__(self, "rectangle", color)
        self.corner = corner
        self.width = width
        self.height = height
```

We can first see that instead of `object` in the class definition, we use `Shape`. The class name in parenthesis defines the IS-A relationship between our new class and some other class. In this case, a `Rectangle` IS-A `Shape`.

The `__init__` method is a little hairier now. First, we've added a `color` parameter so that we can set the color of the shape. The first line within the constructor looks messy, but all we're doing is invoking the constructor of the `Rectangle`'s *parent* class, which is `Shape`. We have to explicitly say `Shape.__init__` to identify the method to call, and we also have to explicitly pass in `self` as the first parameter. This is one of the (very) few situations in which you ever have to invoke a magic method directly.

When we invoke the `Shape` constructor, our object gets outfitted with a `name` and `color`. When we return, we add the `corner`, `width`, and `height` attributes.

Now the fun begins. Let's create a `Rectangle` and manipulate it:

```
>>> r = Rectangle(Point(3,5), 5, 10, "blue")
>>> print r
I am blue rectangle.
```

How did we get such output when we didn't define a `__str__` method in `Rectangle`? Because our `Rectangle` class inherited all the methods of its parent class, `Shape`!

What if we try to get the `perimeter` and `area` for the `Rectangle`?

```
>>> print r.perimeter()
0.0
>>> print r.area()
0.0
```

Since we inherited the methods from `Shape`, we got zeroes. To make our `Rectangle` more useful, what we can do is **override** and redefine how `area` and `perimeter` should work for a rectangle:

```
# inside the Rectangle class definition

    def perimeter(self):
        return self.width * 2 + self.height * 2

    def area(self):
        return self.width * self.height
```

Now, when we ask a rectangle to give us its perimeter and area, it responds appropriately:

```
>>> r = Rectangle(Point(3,5), 5, 10, "blue")
>>> print r.perimeter()
30
>>> print r.area()
50

**Examples**:

1. Refactor the ``Circle`` class so that it inherits from ``Shape``.
```

15.7 Copying objects

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The `copy` module contains a function called `copy` that can duplicate any object:

```
>>> p1 = Point(3.0, 4.0)
>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` and `p2` contain the same data, but they are not the same Point.

```
>>> print p1
Point (3.0, 4.0)
>>> print p2
Point (3.0, 4.0)
>>> p1 is p2
False
>>> p1 == p2
False
```

The `is` operator indicates that `p1` and `p2` are not the same object, which is what we expected. But you might have expected == to yield `True` because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the `is` operator; it checks object identity, not object equivalence. This behavior can be changed—we'll see how later.

If you use `copy.copy` to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

```
>>> import copy
>>> box = Rectangle(Point(3, 2), 5, 10)
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Here is what the object diagram looks like:

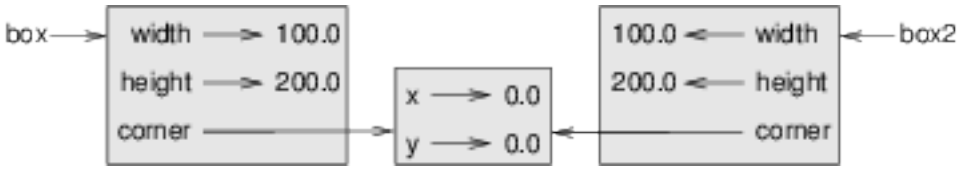


Figure 15.3: Two `rectangle` objects that refer to the same `point` object in memory.

This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.

For most applications, this is not what you want. In this example, invoking `grow_rectangle` on one of the Rectangles would not affect the other, but invoking `move_rectangle` on either would affect both! This behavior is confusing and error-prone.

Fortunately, the `copy` module contains a method named `deepcopy` that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. You will not be surprised to learn that this operation is called a

deep copy.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

`box3` and `box` are completely separate objects.

Example:

1. Write a version of `move_rectangle` that creates and returns a new Rectangle instead of modifying the old one.

15.8 An in-depth example: card games

In this section we will develop classes to represent playing cards, decks of cards, and poker hands. If you don't play poker, you can read about it at http://wikipedia.org/wiki/Poker, but you don't have to; I'll tell you what you need to know for the exercises.

If you are not familiar with Anglo-American playing cards, you can read about them at http://wikipedia.org/wiki/Playing_cards.

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: `rank` and `suit`. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like `'Spade'` for suits and `'Queen'` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. In this context, "encode" means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be "encryption").

For example, this table shows the suits and the corresponding integer codes:

Spades $\mapsto$ 3 Hearts $\mapsto$ 2 Diamonds $\mapsto$ 1 Clubs $\mapsto$ 0

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack $\mapsto$ 11 Queen $\mapsto$ 12 King $\mapsto$ 13

I am using the $\mapsto$ symbol to make it clear that these mappings are not part of the Python program. They are part of the program design, but they don't appear explicitly in the code.

15.8.1 `Card` class

The class definition for `Card` looks like this:

```
class Card(object):
    """represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

As usual, the init method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a Card, you call `Card` with the suit and rank of the card you want.

```
queen_of_diamonds = Card(1, 12)
```

15.8.2 Class attributes

In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits. A natural way to do that is with lists of strings. We assign these lists to **class attributes**:

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

Variables like `suit_names` and `rank_names`, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object `Card`.

This term distinguishes them from variables like `suit` and `rank`, which are called **instance variables** because they are associated with a particular instance.

Both kinds of attribute are accessed using dot notation. For example, in `__str__`, `self` is a Card object, and `self.rank` is its rank. Similarly, `Card` is a class object, and `Card.rank_names` is a list of strings associated with the class.

Every card has its own `suit` and `rank`, but there is only one copy of `suit_names` and `rank_names`.

Putting it all together, the expression `Card.rank_names[self.rank]` means "use the attribute `rank` from the object `self` as an index into the list `rank_names` from the class `Card`, and select the appropriate string."

The first element of `rank_names` is `None` because there is no card with rank zero. By including `None` as a placekeeper, we get a mapping with the nice property that the index 2 maps to the string `'2'`, and so on. To avoid this tweak, we could have used a dictionary instead of a list.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(2, 11)
>>> print card1
Jack of Hearts
```

`Card` is a class object, so it has type `type`. `card1` has type `Card`. (To save space, I didn't draw the contents of `suit_names` and `rank_names`).

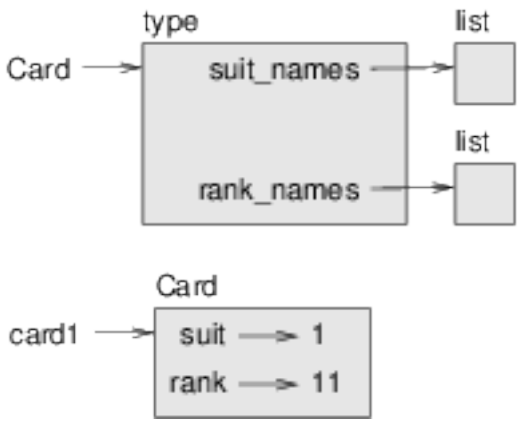


Figure 15.4: Diagram that shows the `Card` class object and one Card instance.

15.8.3 Comparing cards

For built-in types, there are relational operators (<, >, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another. For user-defined types, we can override the behavior of the built-in operators by providing a method named `__cmp__`.

`__cmp__` takes two parameters, `self` and `other`, and returns a positive number if the first object is greater, a negative number if the second object is greater, and 0 if they are equal to each other.

The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

The answer might depend on what game you are playing, but to keep things simple, we'll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

With that decided, we can write `__cmp__`:

```
# inside class Card:

    def __cmp__(self, other):
        # check the suits
        if self.suit > other.suit: return 1
        if self.suit < other.suit: return -1

        # suits are the same... check ranks
        if self.rank > other.rank: return 1
        if self.rank < other.rank: return -1

        # ranks are the same... it's a tie
        return 0
```

You can write this more concisely using tuple comparison:

```
# inside class Card:

    def __cmp__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return cmp(t1, t2)
```

The built-in function `cmp` has the same interface as the method `__cmp__`: it takes two values and returns a positive number if the first is larger, a negative number if the second is larger, and 0 if they are equal.

15.8.4 Decks

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.

The following is a class definition for `Deck`. The init method creates the attribute `cards` and generates the standard set of fifty-two cards:

```
class Deck(object):

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Each iteration creates a new Card with the current suit and rank, and appends it to `self.cards`.

15.8.5 Printing the deck

Here is a `__str__` method for `Deck`:

```
#inside class Deck:

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using `join`. The built-in function `str` invokes the `__str__` method on each card and returns the string representation.

Since we invoke `join` on a newline character, the cards are separated by newlines. Here's what the result looks like:

```
>>> deck = Deck()
>>> print deck
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

15.8.6 Add, remove, shuffle and sort

To deal cards, we would like a method that removes a card from the deck and returns it. The list method `pop` provides a convenient way to do that:

```
#inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

Since `pop` removes the *last* card in the list, we are dealing from the bottom of the deck. In real life bottom dealing is frowned upon [1], but in this context it's ok.

To add a card, we can use the list method `append`:

```
#inside class Deck:

    def add_card(self, card):
        self.cards.append(card)
```

A method like this that uses another function without doing much real work is sometimes called a **veneer**. The metaphor comes from woodworking, where it is common to glue a thin layer of good quality wood to the surface of a cheaper piece of wood.

In this case we are defining a "thin" method that expresses a list operation in terms that are appropriate for decks.

As another example, we can write a Deck method named `shuffle` using the function `shuffle` from the `random` module:

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

Don't forget to import `random`.

Example:

1. Write a Deck method named `sort` that uses the list method `sort` to sort the cards in a `Deck`. `sort` uses the `__cmp__` method we defined to determine sort order.

15.9 `Hand` class

Let's that we now want a class to represent a "hand," that is, the set of cards held by one player. A hand is similar to a deck: both are made up of a set of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don't make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance.

The definition of a child class is like other class definitions, but the name of the parent class appears in parentheses:

```
class Hand(Deck):
    """represents a hand of playing cards"""
```

[1] See http://wikipedia.org/wiki/Bottom_dealing

This definition indicates that `Hand` inherits from `Deck`; that means we can use methods like `pop_card` and `add_card` for Hands as well as Decks.

`Hand` also inherits `__init__` from `Deck`, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the init method for Hands should initialize `cards` with an empty list.

If we provide an init method in the `Hand` class, it overrides the one in the `Deck` class:

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

So when you create a Hand, Python invokes this init method:

```
>>> hand = Hand('new hand')
>>> print hand.cards
[]
>>> print hand.label
new hand
```

But the other methods are inherited from `Deck`, so we can use `pop_card` and `add_card` to deal a card:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print hand
King of Spades
```

A natural next step is to encapsulate this code in a method called `move_cards`:

```
#inside class Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

`move_cards` takes two arguments, a Hand object and the number of cards to deal. It modifies both `self` and `hand`, and returns `None`.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use `move_cards` for any of these operations: `self` can be either a Deck or a Hand, and `hand`, despite the name, can also be a `Deck`.

Example:

1. Write a Deck method called `deal_hands` that takes two parameters, the number of hands and the number of cards per hand, and that creates new Hand objects, deals the appropriate number of cards per hand, and returns a list of Hand objects.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the program easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be scattered among several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

15.10 Class diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point. This kind of relationship is called **HAS-A**, as in, "a Rectangle has a Point."
- One class might inherit from another. This relationship is called **IS-A**, as in, "A Rectangle is a kind of Shape."
- One class might depend on another in the sense that changes in one class would require changes in the other.

A **class diagram** is a graphical representation of these relationships [2]. For example, this diagram shows the relationships between `Card`, `Deck` and `Hand`.

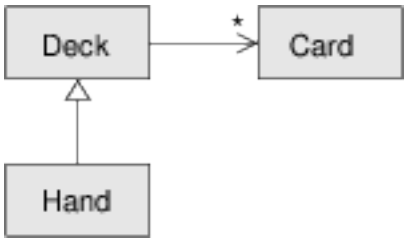


Figure 15.5: Inheritance diagram for `Point`, `Shape`, and `Rectangle`.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Rectangle inherits from Shape.

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

A more detailed diagram might show that a Deck actually contains a *list* of Cards, but built-in types like list and dict are usually not included in class diagrams.

15.11 Debugging

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesn't exist, you get an `AttributeError`:

```
>>> p = Point()
>>> print p.z
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
<type '__main__.Point'>
```

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr`:

[2] The diagrams I am using here are similar to UML (see http://wikipedia.org/wiki/Unified_Modeling_Language), with a few simplifications.

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

The first argument can be any object; the second argument is a *string* that contains the name of the attribute.

It is legal to add attributes to objects at any point in the execution of a program, but if you are a stickler for type theory, it is a dubious practice to have objects of the same type with different attribute sets. It is usually a good idea to initialize all of an objects attributes in the `__init__` method.

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr` (see above).

Another way to access the attributes of an object is through the special attribute `__dict__`, which is a dictionary that maps attribute names (as strings) and values:

```
>>> p = Point(3, 4)
>>> print p.__dict__
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
    for attr in obj.__dict__:
        print attr, getattr(obj, attr)
```

`print_attributes` traverses the items in the object's dictionary and prints each attribute name and its corresponding value.

The built-in function `getattr` takes an object and an attribute name (as a string) and returns the attribute's value.

Inheritance can make debugging a challenge because when you invoke a method on an object, you might not know which method will be invoked.

Suppose you are writing a function that works with Hand objects. You would like it to work with all kinds of Hands, like PokerHands, BridgeHands, etc. If you invoke a method like `shuffle`, you might get the one defined in `Deck`, but if any of the subclasses override this method, you'll get that version instead.

Any time you are unsure about the flow of execution through your program, the simplest solution is to add print statements at the beginning of the relevant methods. If `Deck.shuffle` prints a message that says something like `Running Deck.shuffle`, then as the program runs it traces the flow of execution.

As an alternative, you could use this function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method:

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Here's an example:

```
>>> hand = Hand()
>>> print find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

So the `shuffle` method for this Hand is the one in `Deck`.

`find_defining_class` uses the `mro` method to get the list of class objects (types) that will be searched for methods. "MRO" stands for "method resolution order."

Here's a program design suggestion: whenever you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you obey this rule, you will find that any function designed to work with an instance of a superclass, like a Deck, will also work with instances of subclasses like a Hand or PokerHand.

If you violate this rule, your code will collapse like (sorry) a house of cards.

15.12 Glossary

class: A user-defined type. A class definition creates a new class object.

class object: An object that contains information about a user-defined type. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

attribute: One of the named values associated with an object. Also referred to as *instance variables*.

method: A function that is defined inside a class definition and is invoked on instances of that class.

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

subject: The object a method is invoked on.

constructor: A special method always named `__init__` that handles initializing the values of attributes in an object, and any other setup required when a new instance is created.

magic methods: Method names that begin and end with __; they are implicitly and automatically invoked by the Python interpreter.

object-oriented language: A language that provides features, such as user-defined classes and method syntax, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods. Also referred to as OOP.

embedded (object): An object that is stored as an attribute of another object.

HAS-A relationship: The relationship between two classes where instances of one class contain references to instances of the other.

IS-A relationship: The relationship between a child class and its parent class.

inheritance: The ability to define a new class that is a modified version of a previously defined class.

parent class: The class from which a child class inherits.

child class: A new class created by inheriting from an existing class; also called a "subclass."

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.

class attribute: An attribute associated with a class object. Class attributes are defined inside a class definition but outside any method.

veneer: A method or function that provides a different interface to another function without doing much computation.

class diagram: A diagram that shows the classes in a program and the relationships between them.

15.13 Exercises

1. Write a class definition for a Date object that has attributes `day`, `month` and `year`. Write a function called `increment_date` that takes a Date object, `date` and an integer, `n`, and returns a new Date object that represents the day `n` days after `date`. Hint: "Thirty days hath September..." Challenge: does your function deal with leap years correctly? See http://wikipedia.org/wiki/Leap_year.

2. The built in `datetime` module provides `date` and `time` objects, each with a rich set of methods and operators. Read the documentation at http://docs.python.org/lib/datetime-date.html.

   (a) Use the `datetime` module to write a program that gets the current date and prints the day of the week.

   (b) Write a program that takes a birthday as input and prints the user's age and the number of days, hours, minutes and seconds until their next birthday.

3. Write a definition for a class named `Kangaroo` with the following methods:

   (a) An `__init__` method that initializes an attribute named `pouch_contents` to an empty list.

   (b) A method named `put_in_pouch` that takes an object of any type and adds it to `pouch_contents`.

   (c) A `__str__` method that returns a string representation of the Kangaroo object and the contents of the pouch.

   Test your code by creating two `Kangaroo` objects, assigning them to variables named `kanga` and `roo`, and then adding `roo` to the contents of `kanga`'s pouch.

4. The following code is a solution to the previous problem, except that it contains a nasty bug. Find, describe, and fix the problem.

```
class Kangaroo(object):
    """a Kangaroo is a marsupial"""

    def __init__(self, contents=[]):
        """initialize the pouch contents; the default value is
        an empty list"""
        self.pouch_contents = contents

    def __str__(self):
        """return a string representaion of this Kangaroo and
        the contents of the pouch, with one item per line"""
        t = [ object.__str__(self) + ' with pouch contents:' ]
        for obj in self.pouch_contents:
            s = '    ' + object.__str__(obj)
            t.append(s)
        return '\n'.join(t)

    def put_in_pouch(self, item):
        """add a new item to the pouch contents"""
        self.pouch_contents.append(item)

kanga = Kangaroo()
roo = Kangaroo()
kanga.put_in_pouch('wallet')
kanga.put_in_pouch('car keys')
kanga.put_in_pouch(roo)

print kanga
```

```
# If you run this program as is, it seems to work.
# To see the problem, trying printing roo.
```

5. The table below shows possible hands in poker, in increasing order of value (and decreasing order of probability):

pair	two cards with the same rank
two pair	two pairs of cards with the same rank
three of a kind	three cards with the same rank
straight	five cards with ranks in sequence (aces can be high or low, so `Ace-2-3-4-5` is a straight and so is `10-Jack-Queen-King-Ace`, but `Queen-King-Ace-2-3` is not.)
flush	five cards with the same suit
full house	three cards with one rank, two cards with another
four of a kind	four cards with the same rank
straight flush	five cards in sequence (as defined above) and with the same suit

The goal of these exercises is to estimate the probability of drawing these various hands.

(a) Using the `Card`, `Hand`, and `Deck` classes created in this chapter, create a `PokerHand` class that can hold up to 7 cards at once.

(b) Write a `main` function that deals cards from a `Deck` object and adds them to a `PokerHand` object.

(c) Write a `isStraightFlush` method for the `PokerHand` class that tests whether the hand contains a straight flush.

(d) Add methods to `PokerHand` named `has_pair`, `has_twopair`, etc. that return True or False according to whether or not the hand meets the relevant criteria. Your code should work correctly for "hands" that contain any number of cards (although 5 and 7 are the most common sizes).

(e) Write a method named `classify` that figures out the highest-value classification for a hand and sets the `label` attribute accordingly. For example, a 7-card hand might contain a flush and a pair; it should be labeled "flush".

(f) When you are convinced that your classification methods are working, the next step is to estimate the probabilities of the various hands. Write a function that shuffles a deck of cards, divides it into hands, classifies the hands, and counts the number of times various classifications appear.

(g) Print a table of the classifications and their probabilities. Run your program with larger and larger numbers of hands until the output values converge to a reasonable degree of accuracy. Compare your results to the values at http://wikipedia.org/wiki/Hand_rankings.

6. This exercise uses the `turtle` module. You will write code that makes Turtles play tag. If you are not familiar with the rules of tag, see http://wikipedia.org/wiki/Tag_(game).

(a) Type in the following code and run it. You should see a turtle screen with three turtles that start wandering around the screen at random.

```
'''
Wobbler class originally written by Allen Downey.
Modified by J. Sommers for use with vanilla turtle rather
than TurtleWorld.
'''

import turtle
import random
```

```
class Wobbler(turtle.Turtle):
    """a Wobbler is a kind of turtle with attributes for speed and
    clumsiness."""

    def __init__(self, speed=1, clumsiness=60, color='red'):
        turtle.Turtle.__init__(self)
        self.delay = 0
        self.speed = speed
        self.clumsiness = clumsiness
        self.pencolor(color)

        # move to the starting position
        self.penup()
        self.right(random.randint(0,360))
        self.backward(150)
        self.pendown()

    def step(self):
        """step is invoked by the timer function on every Wobbler, once
        per time step."""

        self.steer()
        self.wobble()
        self.move()

    def move(self):
        """move forward in proportion to self.speed"""
        self.forward(self.speed)

    def wobble(self):
        """make a random turn in proportion to self.clumsiness"""
        dir = random.randint(0,self.clumsiness) - random.randint(0,self.clumsiness)
        self.right(dir)

    def steer(self):
        """steer the Wobbler in the general direction it should go.
        Postcondition: the Wobbler's heading may be changed, but
        its position may not."""
        self.right(10)


def timerfunction():
    for t in turtle.turtles():
        t.step()
    turtle.ontimer(timerfunction, 100)


if __name__ == '__main__':

    # make 3 turtles
    turtle_colors = ['red','blue','yellow']
    i = 1.0
    for i in range(3):
        w = Wobbler(i, i*30, turtle_colors[i])
        i += 0.5

    timerfunction()
    turtle.mainloop()
```

(b) Read the code and make sure you understand how it works. The `Wobbler` class inherits from `Turtle`, which means that the `Turtle` methods `left`, `right`, `forward` and `backward` work on Wobblers.

The `step` method gets invoked by the `timerfunction`. It invokes `steer`, which turns the Turtle in the desired direction, `wobble`, which makes a random turn in proportion to the Turtle's clumsiness, and `move`, which moves forward a few pixels, depending on the Turtle's speed.

(c) Create a class named `Tagger` that inherits from `Wobbler`. Change the call in `main` to invoke `Tagger` instead of `Wobbler` when creating the turtles.

(d) Add a `steer` method to `Tagger` to override the one in `Wobbler`. As a starting place, write a version that always points the Turtle toward the origin. Hint: use the math function `atan2` and the Turtle attributes `x`, `y` and `heading`.

(e) Modify `steer` so that the Turtles stay on the screen.

(f) Modify `steer` so that each Turtle points toward its nearest neighbor. Hint: Turtles have an attribute, `screen`, that is a reference to the `Screen` they live in, and `Screen` has a method `turtles` that returns a list of all the `Turtle` objects on the screen.

(g) Modify `steer` so the Turtles play tag. You can add methods to `Tagger` and you can override `steer` and `__init__`, but you may not modify or override `step`, `wobble` or `move`. Also, `steer` is allowed to change the heading of the Turtle but not the position.

Adjust the rules and your `steer` method for good quality play; for example, it should be possible for the slow Turtle to tag the faster Turtles eventually.

CHAPTER 16

Appendix: Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

- Syntax errors are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program. Example: Omitting the colon at the end of a `def` statement yields the somewhat redundant message `SyntaxError: invalid syntax`.
- Runtime errors are produced by the interpreter if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes the runtime error "maximum recursion depth exceeded."
- Semantic errors are problems with a program that runs without producing error messages but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an incorrect result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

16.1 Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where Python noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.
2. Check that you have a colon at the end of the header of every compound statement, including `for`, `while`, `if`, and `def` statements.

3. Make sure that any strings in the code have matching quotation marks.
4. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an `invalid token` error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
5. An unclosed opening operator—`(`, `{`, or `[`—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
6. Check for the classic = instead of == inside a conditional.
7. Check the indentation to make sure it lines up the way it is supposed to. Python can handle space and tabs, but if you mix them it can cause problems. The best way to avoid this problem is to use a text editor that knows about Python and generates consistent indentation.

If nothing works, move on to the next section...

16.1.1 I keep making changes and it makes no difference.

If the interpreter says there is an error and you don't see it, that might be because you and the interpreter are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one Python is trying to run.

If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run it again. If the interpreter doesn't find the new error, you are not running the new code.

There are a few likely culprits:

- You edited the file and forgot to save the changes before running it again. Some programming environments do this for you, but some don't.
- You changed the name of the file, but you are still running the old name.
- Something in your development environment is configured incorrectly.
- If you are writing a module and using `import`, make sure you don't give your module the same name as one of the standard Python modules.
- If you are using `import` to read a module, remember that you have to restart the interpreter or use `reload` to read a modified file. If you import the module again, it doesn't do anything.

If you get stuck and you can't figure out what is going on, one approach is to start again with a new program like "Hello, World!," and make sure you can get a known program to run. Then gradually add the pieces of the original program to the new one.

16.2 Runtime errors

Once your program is syntactically correct, Python can compile it and at least start running it. What could possibly go wrong?

16.2.1 My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the "Flow of Execution" section below.

16.2.2 My program hangs.

If a program stops and seems to be doing nothing, it is "hanging." Often that means that it is caught in an infinite loop or infinite recursion.

- If there is a particular loop that you suspect is the problem, add a `print` statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop."

  Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the "Infinite Loop" section below.

- Most of the time, an infinite recursion will cause the program to run for a while and then produce a "RuntimeError: Maximum recursion depth exceeded" error. If that happens, go to the "Infinite Recursion" section below.

  If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the "Infinite Recursion" section.

- If neither of those steps works, start testing other loops and other recursive functions and methods.

- If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the "Flow of Execution" section below.

Infinite Loop

If you think you have an infinite loop and you think you know what loop is causing the problem, add a `print` statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

For example:

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print "x: ", x
    print "y: ", y
    print "condition: ", (x > 0 and y < 0)
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be `false`. If the loop keeps going, you will be able to see the values of `x` and `y`, and you might figure out why they are not being updated correctly.

Infinite Recursion

Most of the time, an infinite recursion will cause the program to run for a while and then produce a `Maximum recursion depth exceeded` error.

If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the function or method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a `print` statement at the beginning of the function or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

16.3 Flow of Execution

If you are not sure how the flow of execution is moving through your program, add `print` statements to the beginning of each function with a message like "entering function `foo`," where `foo` is the name of the function.

Now when you run the program, it will print a trace of each function as it is invoked.

16.3.1 When I run the program I get an exception.

If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.

The traceback identifies the function that is currently running, and then the function that invoked it, and then the function that invoked *that*, and so on. In other words, it traces the sequence of function invocations that got you to where you are. It also includes the line number in your file where each of these calls occurs.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

NameError: You are trying to use a variable that doesn't exist in the current environment. Remember that local variables are local. You cannot refer to them from outside the function where they are defined.

TypeError: There are several possible causes:

- You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.
- There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is called for.
- You are passing the wrong number of arguments to a function or method. For methods, look at the method definition and check that the first parameter is `self`. Then look at the method invocation; make sure you are invoking the method on an object with the right type and providing the other arguments correctly.

KeyError: You are trying to access an element of a dictionary using a key that the dictionary does not contain.

AttributeError: You are trying to access an attribute or method that does not exist. Check the spelling! You can use `dir` to list the attributes that do exist.

If an AttributeError indicates that an object has `NoneType`, that means that it is `None`. One common cause is forgetting to return a value from a function; if you get to the end of a function without hitting a `return` statement, it returns `None`. Another common cause is using the result from a list method, like `sort`, that returns `None`.

IndexError: The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a `print` statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

The Python debugger (`pdb`) is useful for tracking down Exceptions because it allows you to examine the state of the program immediately before the error. You can read about `pdb` at http://docs.python.org/lib/module-pdb.html.

16.3.2 I added so many `print` statements I get inundated with output.

One of the problems with using `print` statements for debugging is that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out `print` statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are searching a list, search a *small* list. If the program takes input from the user, give it the simplest input that causes the problem.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smaller functions and testing them separately.

Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think shouldn't affect the program, and it does, that can tip you off.

16.4 Semantic errors

In some ways, semantic errors are the hardest to debug, because the interpreter provides no information about what is wrong. Only you know what the program is supposed to do.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast.

You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed `print` statements is often short compared to setting up the debugger, inserting and removing breakpoints, and "stepping" the program to where the error is occurring.

16.4.1 My program doesn't work.

You should ask yourself these questions:

- Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to functions or methods in other Python modules. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

16.4.2 I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

This can be rewritten as:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression $\frac{x}{2\pi}$ into Python, you might write:

```
y = x / 2 * math.pi
```

That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes $x\pi/2$.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
y = x / (2 * math.pi)
```

Whenever you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the rules of precedence.

16.4.3 I've got a function or method that doesn't return what I expect.

If you have a `return` statement with a complex expression, you don't have a chance to print the `return` value before returning. Again, you can use a temporary variable. For example, instead of:

```
return self.hands[i].removeMatches()
```

you could write:

```
count = self.hands[i].removeMatches()
return count
```

Now you have the opportunity to display the value of `count` before returning.

16.4.4 I'm really, really stuck and I need help.

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these symptoms:

- Frustration and rage.
- Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backward").
- Random walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. I often find bugs when I am away from the computer and let my mind wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

16.4.5 No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you are prepared. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have `print` statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need:

- If there is an error message, what is it and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, the goal is not just to make the program work. The goal is to learn *how* to make the program work.

CHAPTER 17

Postscript

Note: this was originally the preface to "How to think like a computer scientist" by Allen Downey.

17.1 The strange history of this book

In January 1999 I was preparing to teach an introductory programming class in Java. I had taught it three times and I was getting frustrated. The failure rate in the class was too high and, even for students who succeeded, the overall level of achievement was too low.

One of the problems I saw was the books. They were too big, with too much unnecessary detail about Java, and not enough high-level guidance about how to program. And they all suffered from the trap door effect: they would start out easy, proceed gradually, and then somewhere around Chapter 5 the bottom would fall out. The students would get too much new material, too fast, and I would spend the rest of the semester picking up the pieces.

Two weeks before the first day of classes, I decided to write my own book. My goals were:

- Keep it short. It is better for students to read 10 pages than not read 50 pages.
- Be careful with vocabulary. I tried to minimize the jargon and define each term at first use.
- Build gradually. To avoid trap doors, I took the most difficult topics and split them into a series of small steps.
- Focus on programming, not the programming language. I included the minimum useful subset of Java and left out the rest.

I needed a title, so on a whim I chose *How to Think Like a Computer Scientist*.

My first version was rough, but it worked. Students did the reading, and they understood enough that I could spend class time on the hard topics, the interesting topics and (most important) letting the students practice.

I released the book under the GNU Free Documentation License, which allows users to copy, modify, and distribute the book.

What happened next is the cool part. Jeff Elkner, a high school teacher in Virginia, adopted my book and translated it into Python. He sent me a copy of his translation, and I had the unusual experience of learning Python by reading my own book.

Jeff and I revised the book, incorporated a case study by Chris Meyers, and in 2001 we released *How to Think Like a Computer Scientist: Learning with Python*, also under the GNU Free Documentation License. As Green Tea Press, I published the book and started selling hard copies through Amazon.com and college book stores. Other books from Green Tea Press are available at http://greenteapress.com.

In 2003 I started teaching at Olin College and I got to teach Python for the first time. The contrast with Java was striking. Students struggled less, learned more, worked on more interesting projects, and generally had a lot more fun.

Over the last five years I have continued to develop the book, correcting errors, improving some of the examples and adding material, especially exercises. In 2008 I started work on a major revision—at the same time, I was contacted by an editor at Cambridge University Press who was interested in publishing the next edition. Good timing!

The result is this book, now with the less grandiose title *Think Python*. Some of the changes are:

- I added a section about debugging at the end of each chapter. These sections present general techniques for finding and avoiding bugs, and warnings about Python pitfalls.
- I removed the material in the last few chapters about the implementation of lists and trees. I still love those topics, but I thought they were incongruent with the rest of the book.
- I added more exercises, ranging from short tests of understanding to a few substantial projects.
- I added a series of case studies—longer examples with exercises, solutions, and discussion. Some of them are based on Swampy, a suite of Python programs I wrote for use in my classes. Swampy, code examples, and some solutions are available from http://thinkpython.com.
- I expanded the discussion of program development plans and basic design patterns.
- The use of Python is more idiomatic. The book is still about programming, not Python, but now I think the book gets more leverage from the language.

I hope you enjoy working with this book, and that it helps you learn to program and think, at least a little bit, like a computer scientist.

Allen B. Downey Needham MA Allen Downey is an Associate Professor of Computer Science at the Franklin W. Olin College of Engineering.

17.2 Acknowledgements

First and most importantly, I thank Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language.

I also thank Chris Meyers, who contributed several sections to *How to Think Like a Computer Scientist*.

And I thank the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible.

I also thank the editors at Lulu who worked on *How to Think Like a Computer Scientist*.

I thank all the students who worked with earlier versions of this book and all the contributors (listed below) who sent in corrections and suggestions.

And I thank my wife, Lisa, for her work on this book, and Green Tea Press, and everything else, too.

17.3 Contributor List

More than 100 sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

If you have a suggestion or correction, please send email to `feedback@thinkpython.com`. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam 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 and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word ‘‘unconsciously” in Chapter 1 needed to be changed to ‘‘subconsciously”.
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the `increment` function in Chapter 13.
- John Ouzts corrected the definition of ‘‘return value” in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the printTime function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen’s Perl script that generates HTML from LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka 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.

- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between *gleich* and *selbe*.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.
- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and helped us update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.
- Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice typo.
- D. J. Webre suggested a clarification in Chapter 3.
- Ken found a fistful of errors in Chapters 8, 9 and 11.
- Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.
- Curtis Yanko suggested a clarification in Chapter 2.
- Ben Logan sent in a number of typos and problems with translating the book into HTML.
- Jason Armstrong saw the missing word in Chapter 2.
- Louis Cordier noticed a spot in Chapter 16 where the code didn't match the text.
- Brian Cain suggested several clarifications in Chapters 2 and 3.
- Rob Black sent in a passel of corrections, including some changes for Python 2.2.
- Jean-Philippe Rey at Ecole Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.
- Jason Mader at George Washington University made a number of useful suggestions and corrections.
- Jan Gundtofte-Bruun reminded us that "a error" is an error.
- Abel David and Alexis Dinno reminded us that the plural of "matrix" is "matrices", not "matrixes". This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.
- Charles Thayer encouraged us to get rid of the semi-colons we had put at the ends of some statements and to clean up our use of "argument" and "parameter".
- Roger Sperberg pointed out a twisted piece of logic in Chapter 3.
- Sam Bull pointed out a confusing paragraph in Chapter 2.
- Andrew Cheung pointed out two instances of "use before def."
- C. Corey Capel spotted the missing word in the Third Theorem of Debugging and a typo in Chapter 4.

- Alessandra helped clear up some Turtle confusion.
- Wim Champagne found a brain-o in a dictionary example.
- Douglas Wright pointed out a problem with floor division in `arc`.
- Jared Spindor found some jetsam at the end of a sentence.
- Lin Peiheng sent a number of very helpful suggestions.
- Ray Hagtvedt sent in two errors and a not-quite-error.
- Torsten Hübsch pointed out an inconsistency in Swampy.
- Inga Petuhhov corrected an example in Chapter 14.
- Arne Babenhauserheide sent several helpful corrections.
- Mark E. Casida is is good at spotting repeated words.
- Scott Tyler filled in a that was missing. And then sent in a heap of corrections.
- Gordon Shephard sent in several corrections, all in separate emails.
- Andrew Turner `spot`ted an error in Chapter 8.
- Adam Hobart fixed a problem with floor division in `arc`.
- Daryl Hammond and Sarah Zimmerman pointed out that I served up `math.pi` too early. And Zim spotted a typo.
- George Sass found a bug in a Debugging section.
- Brian Bingham suggested Exercise {exrotatepairs}.
- Leah Engelbert-Fenton pointed out that I used `tuple` as a variable name, contrary to my own advice. And then found a bunch of typos and a "use before def."
- Joe Funke spotted a typo.
- Chao-chao Chen found an inconsistency in the Fibonacci example.
- Jeff Paine knows the difference between space and spam.
- Lubos Pintes sent in a typo.
- Gregg Lind and Abigail Heithoff suggested Exercise {checksum}.
- Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary * Concrete Abstractions*, which you might want to read when you are done with this book.
- Chotipat Pornavalai found an error in an error message.
- Stanislaw Antol sent a list of very helpful suggestions.
- Eric Pashman sent a number of corrections for Chapters 4–11.
- Miguel Azevedo found some typos.
- Jianhua Liu sent in a long list of corrections.
- Nick King found a missing word.
- Martin Zuther sent a long list of suggestions.
- Adam Zimmerman found an inconsistency in my instance of an "instance" and several other errors.
- Ratnakar Tiwari suggested a footnote explaining degenerate triangles.

- Anurag Goel suggested another solution for `is_abecedarian` and sent some additional corrections. And he knows how to spell Jane Austen.
- Kelli Kratzer spotted one of the typos.
- Mark Griffiths pointed out a confusing example in Chapter 3.
- Roydan Ongie found an error in my Newton's method.
- Patryk Wolowiec helped me with a problem in the HTML version.
- Mark Chonofsky told me about a new keyword in Python 3.0.
- Russell Coleman helped me with my geometry.
- Wei Huang spotted several typographical errors.
- Karen Barber spotted the the oldest typo in the book.
- Nam Nguyen found a typo and pointed out that I used the Decorator pattern but didn't mention it by name.
- Stéphane Morin sent in several corrections and suggestions.
- Paul Stoop corrected a typo in `uses_only`.
- Eric Bronner pointed out a confusion in the discussion of the order of operations.
- Alexandros Gezerlis set a new standard for the number and quality of suggestions he submitted. We are deeply grateful!
- Gray Thomas knows his right from his left.
- Giovanni Escobar Sosa sent a long list of corrections and suggestions.
- Alix Etienne fixed one of the URLs.

Part II

Indices and tables

- *genindex*
- *modindex*
- *search*