Python for Informatics

Exploring Information

Version 0.0.1

Python for Informatics

Exploring Information

Version 0.0.1

Charles Severance

Printing history:

December 2009: Begin to re-mix *Think Python: How to Think Like a Computer Scientist* to produce *Python for Informatics: Exploring Information*

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

April 2002: First edition of *How to Think Like a Computer Scientist*.

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

The LaTeX source for the *Think Python* version of this book is available from `http://www.thinkpython.com`.

The LaTeX source for the *Python for Informatics* version of the book is available (for the moment) from `http://source.sakaiproject.org/contrib//csev/trunk/pyinf/`.

Preface

Python for Informatics: Remixing an Open Book

It is quite natural for academics who are continuously told to ”publish or perish” to want to always create something from scratch that is their own fresh creation. This book is an experiment in not starting from scratch, but instead ”re-mixing” the book titled *Think Python: How to Think Like a Computer Scientist* written by Allen B. Downey, Jeff Elkner and others.

In December of 2009, I was preparing to teach **SI502 - Networked Programming** at the University of Michigan for the fifth semester in a row and decided it was time to write a Python textbook that focused on exploring data instead of understanding algorithms and abstractions. My goal in SI502 is to teach people life-long data handling skills using Python. Few of my students were planning to be be professional computer programmers - instead, they planned be librarians, managers, lawyers, biologists, economists, etc. who happenned to want to skillfully use technology in their chosen field.

I never seemed to find the perfect data-oriented Python book for my course so I set out to write just such a book. Luckily at a faculty meeting three weeks before I was about to start my new book from scratch over the holiday break, Dr. Atul Prakash showed me the *Think Python* book which he had used to teach his Python course that semester.

Think Python follows the structure of most introductory Computer Science textbooks. It is a well-written text with a focus on short, direct explanations and ease of learning.

The book was available under the GNU Free Documentation License so I had all the permission I needed to use the book as the base material for my book. As a courtesy, I sent a note to Allen and Cambridge Press letting them know what I was about to do to their book.

I expect that by the time I am done with *Python for Informatics* over fifty percent of the book will be new. The overall structure will be changed to get to doing data analysis problems as quickly as possible and have a series of running examples and exercises about data analysis. Then I will add chapters on regular expressions, data visualization, working with spreadsheet data, structured query launguage using SQLite, web scraping, and calling REST-based Application Program Interfaces.

The ultimate goal in the shift from a Computer Science to an Informatics focus is to pull topics into the first programming class that can be applied even if one chooses not to become a professional programmer.

What is interesting even with this change of focus is how much of the original *Think Python* book material is directly relevant to this book and how much will fit right into *Python for Informatics* with virtually no change.

By starting with the *Think Python* book, I don't have to write the basic descriptions of the Python language or how to debug programs and instead focus on the topical material that is the value-add of *Python for Informatics*.

Students who find this book interesting and want to further explore a career as a professional programmer should probably look at the *Think Python* book. Because there is a lot of overlap between the two books, you will quickly pick up skills in the additional areas of Computer Science which are covered in *Think Python*. And given that the books have a similar writing style and at times have identical text and examples, you should be able to pick up these new topics with a minimum of effort.

I hope that this book serves an example of why open materials are so important to the future of education, and want to thank Allen B. Downey and Cambridge University Press for their forward looking decision to make the book available under an open Copyright. I hope they are pleased with the results of my efforts and I hope that you the reader are pleased with *our* collective efforts.

Charles Severance
www.dr-chuck.com
December 19, 2009

Charles Severance is a Clinical Assistant Professor at the University of Michigan School of Information.

Preface for "Think Python"

The strange history of "Think Python"

(Allen B. Downey)

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 `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!

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.

Acknowledgements for "Think Python"

(Allen B. Downey)

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 in an Appendix) 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.

Allen B. Downey
Needham MA

Allen Downey is an Associate Professor of Computer Science at the Franklin W. Olin College of Engineering.

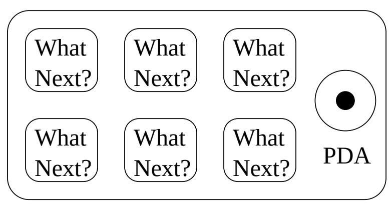
Contents

Chapter 1

Why should you learn to write programs?

Writing programs (or programming) is a very creative and rewarding activity. You can write programs for many reasons ranging from making your living to solving a difficult data analysis problem to having fun to helping someone else solve a problem. This book assumes that *everyone* needs to know how to program and that once you know how to program, you will figure out what you want to do with your newfound skills.

We are surrounded in our daily lives with computers ranging from laptops to cell phones. We can think of these computers as our "personal assistants" who can take care of many things on our behalf. The hardware in our current-day computers is essentially built to continuously ask us the question, "What would you like me to do next?".



Our computers are fast and have vasts amounts of memory and could be very helpful to us if we only knew the language to speak to explain to the computer what we would like it to "do next". If we knew this language we could tell the computer to do tasks on our behalf that were reptitive. Interestingly, the kinds of things computers can do best are often the kinds of things that we humans find boring and mind-numbing.

For example, look at the first three paragraphs of this chapter and tell me the most commonly used word and how many times the word is used. While you were able to read and understand the words in a few seconds, counting them is almost painful because is is not the kind of problem that human minds are designed to solve. For a computer the opposite is true, reading and understanding text from a piece of paper is hard for a computer to do but counting the words and telling you how many times the most used word was used is very easy for the computer:

```
python words.py
Enter file:words.txt
to 16
```

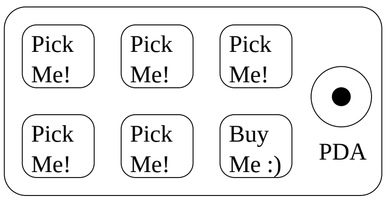
Our "personal information analysis assistant" quickly told us that the word "to" was used sixteen times in the first three paragraphs of this chapter.

It is this very fact that computers are good at things that humans are not so good at is why you need to become skilled at talking "computer language". Once you learn this new language, you can delegate mundane tasks to your partner (the computer), leaving more time for you to do the things that you are uniquely suited for. You bring creativity, intuition, and inventiveness to this partnership.

1.1 Creativity and motivation

While this book is not intended for professional programmers, professional programming can be a very rewarding job both financially and personally. Building useful, elegant, and clever programs for others to use is a very creative activity. Your computer or Personal Digital Assistant (PDA) usually contains many different programs from many different groups of programmers, each competing for your attention and interest. They try their best to meet your needs and give you a great user experience in the process. In some situations, when you choose a piece of software, the programmers are directly compensated because of your choice.

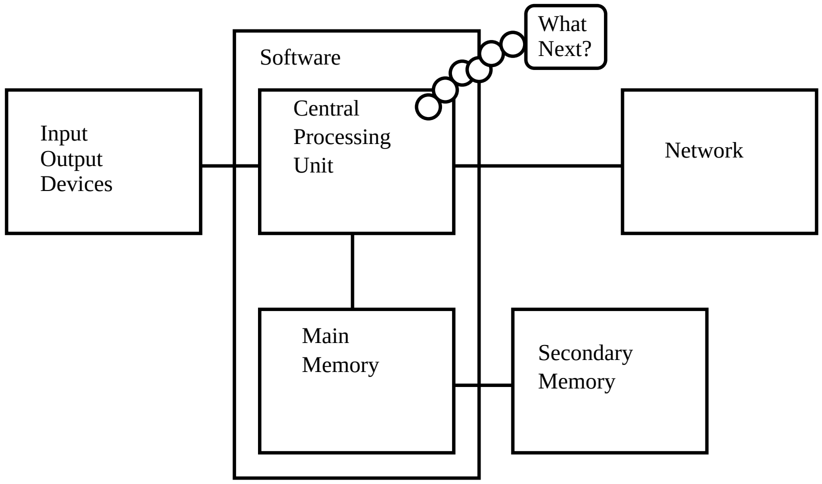
If we think of programs as the creative output of groups of programmers, perhaps the following figure is a more sensible version of our PDA:



For now, our primary motivation is not to make money or please end-users, but instead for us to be more productive in handling the data and information that we will encounter in our lives. When you first start out, you will be both the programmer and end-user of your programs. As you gain skill as a programmer and programming feels more creative to you, your thoughts may turn toward developing programs for others.

1.2 Computer hardware architecture

Before we start out learning the language we speak to give instructions to computers to develop software, we need to learn about more about how computers are built. If you were to take apart your computer or cell phone and look deep inside, you would find the following parts:



The high-level definitions of these parts are as follows:

- The **Central Processing Unit** (or CPU) is that part of the computer that is built to be obsessed with "what is next?". If your computer is rated at 3.0 Gigahertz, it means that the CPU will ask "What next?" three billion times per second. You are going to have to learn how to talk fast to keep up with the CPU.

- The **Main Memory** is used to store information that the CPU needs in a hurry. The main memory is nearly as fast as the CPU. But the information stored in the main memory vanishes when the computer is turned off.

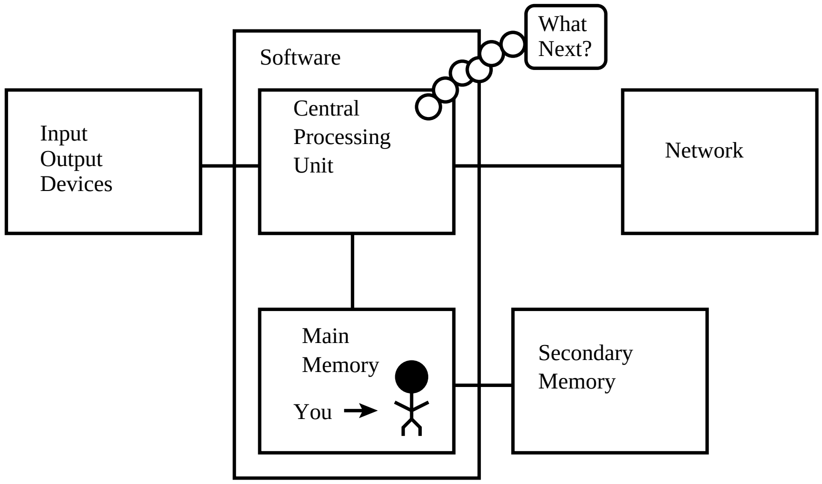
- The **Secondary Memory** is also used to store information, but it is much slower than the main memory. The advantage of the secondary memory is that it can store information even when there is no power to the computer. Examples of secondary memory are disk drives or flash memory (typically found in USB sticks and portable music players).

- The **Input and Output Devices** are simply our screen, keyboard, mouse, microphone, speaker, touchpad, etc. They are all of the ways we interact with the computer.

- These days, most computers also have a **Network Connection** to retrieve information over a network. We can think of the network as a very slow place to store and retrieve data that might now always be "up". So in a sense, the network is a slower and at times unreliable form of **Secondary Memory**

While most of the detail of how these components work is best left to computer builders, it helps to have a some terminology so we can talk about these different parts as we write our programs.

As a programmer, your job is to use and orchestrate each of these resources to solve the problem that you need solving and analyze the data you need analysed. As a programmer you will mostly be "talking" to the CPU and telling it what to do next. Sometimes you will tell the CPU to use the main memory, secondary memory, network, or the input/output devices.



You need to be the person who answers the CPU's "What next?" question. But it would be very uncomfortable to shrink you down to 5mm tall and insert you into the computer just so you could issue a command three billion times per second. So instead, you must write down your instructions in advance. We call these stored instructions a **program** and the act of writing these instructions down and getting the instructions to be correct **programming**.

1.3 Understanding programming

In the rest of this book, we will try to turn you into a person who is skilled in the art of programming. In the end you will be a **programmer** - perhaps not a professional programmer - but at least you will have the skills to look at a data/information analysis problem and develop a program to solve the problem.

In a sense, you need two skills to be a programmer:

- First you need to know the programming language (Python) - you need to know the vocabulary and the grammar. You need to be able spell the words in this new language properly and how to construct well-formed "sentances" in this new languages.
- Second you need to to "tell a story". In writing a story, you combine words and sentances to convey an idea to the reader. There is a skill and art in constructing the story and skill in story writing is improved by doing some writing and getting some feedback. In programming, our program is the "story" and the problem you are trying to solve is the "idea".

Once you learn one programming language such as Python, you will find it much easier to learn a second programming language such as JavaScript or C++. The new programming language has very different vocabulary and grammar but once you learn problem solving skills, they will be the same across all programming languages.

You will learn the "vocabulary" and "sentances" of Python pretty quickly. It will take longer for you to be able to write a coherent program to solve a brand new problem. We teach programming much like we teach writing. We start reading and explaining programs and then we write simple programs and then write increasingly complex programs over time. At some point you "get your muse" and see the patterns on your own and can see more naturally how to take a problem and write a program that solves that problem. And once you get to that point, programming becomes a very pleasant and creative process.

We start with the vocabulary and structure of Python programs. Be patient as the simple examples remind you of when you started reading for the first time.

1.4 The Python programming language

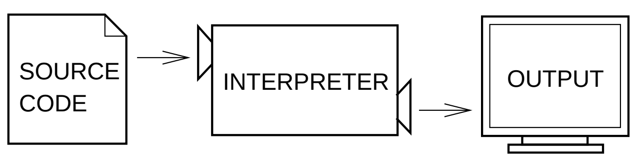
The programming language you will learn is Python. Python is an example of a **high-level language**; other high-level languages you might have heard of are C, C++, Perl, Java, Ruby, and JavaScript. At times, we will include a few examples of the JavaScript language to help cement the basic grammar ideas using two different "vocabularies".

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.

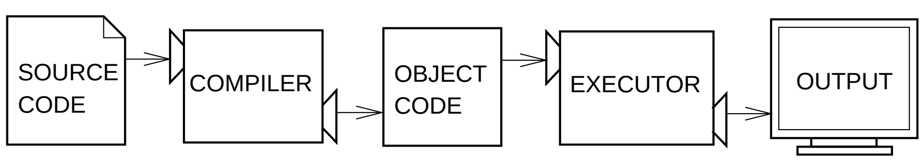
The advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can 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**, **machine code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation.



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`. The chevron is the Python interpreter's way of asking you, "What do you want me to do next?". You will notice that as soon as Python finishes one statement - it immediately is ready for you to type another statement.

Typing commands into the Python interpreter is a great way to experiment with Python's features, but it is a bad way to type in many commands to solve a more complex problem. When we want to write a program, we use a text editor to write the Python instructions into a file, which is called a **script**. By convention, Python scripts have names that end with `.py`.

To execute the script, you have to tell the interpreter the name of the file. In a UNIX command window, you would type `python dinsdale.py`. In other development environments, the details of executing scripts are different. You can find instructions for your environment at the Python Website `python.org`.

Working in interactive mode is convenient for testing small pieces of code because you can type and execute them immediately. But for anything more than a few lines, you should save your code as a script so you can modify and execute it in the future.

1.5 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, pausing if necessary.

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

sequential execution: Perform statements one after another in the order they are encountered in the script.

conditional execution: Check for certain conditions and execute or skip a sequence of statements.

repeated execution: Perform some set of statements repeatedly, usually with some variation.

reuse: Write a set of instructions once and give them a name and then reuse those instructions as needed throuhgout your program.,

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**.

1.6 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**.

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.

1.6.1 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.

1.6.2 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.

1.6.3 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 but not what you meant for it to do.

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

1.6.4 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’ Guide* Beta Version 1).

Later chapters will make more suggestions about debugging and other programming practices.

1.7 Building “sentances” in Python

The rules (or grammar) of Python are simpler and more precise than the rules of a natural language that we use to speak and write.

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, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement; that is, the way the tokens are arranged. The statement $3+=3\$6$ 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.

Exercise 1.1 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.

1.8 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:

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

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

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.

1.9 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.

Whenever you are experimenting with a new feature, you should try to make mistakes. For example, 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!

1.10 Glossary

central processing unit: The heart of any computer. It is what runs the software that we write; also called "CPU" or "the processor".

[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*.

main memory: Stores programs and data. Main memory loses its information when the power is turned off.

secondary memory: Stores programs and data and retains its information even when the power is turned off. Generally slower than main memory. Examples of secondary memory include disk drives and flash member in USB sticks.

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 code" or "assembly language."

machine code: The lowest level language for software which is the language that is directly executed by the central processing unit (CPU).

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.

1.11 Exercises

Exercise 1.2 Use a web browser to go to the Python Website `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.

Exercise 1.3 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.

If this example doesn't work, you may need to install additional Python documentation or set an environment variable; the details depend on your operating system and version of Python.

Exercise 1.4 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 `*`.

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).

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 `1`, `2`, and `'Hello, World!'`.

These values belong to different **types**: `2` is an integer, and `'Hello, World!'` is a **string**, so-called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works 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 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

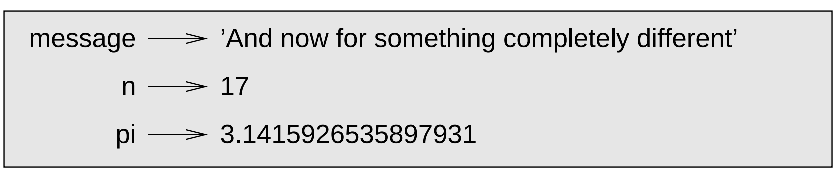
One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

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:



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'>
```

Exercise 2.1 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 (you'll see why 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 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 a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

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

produces the output

```
1
2
```

The assignment statement produces no 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. I won't cover bitwise operators in this book, but you can read about them at `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.

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
```

[2] In Python 3.0, the result of this division is a `float`. The new operator `//` performs integer division.

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! This is a common source of confusion for beginners.

Exercise 2.2 Type the following statements in the Python interpreter to see what they do:

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

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 reorder the operands or use parentheses.

2.8 Modulus operator

The **modulus 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.

2.9 String operations

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 linking 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?

2.10 Keyboard input

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called `raw_input` that gets input from the keyboard[3]. 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.

[3] In Python 3.0, this function is named `input`.

```
>>> input = raw_input()
What are you waiting for?
>>> print input
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` using the `int()` function:

```
>>> 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.

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 mnemonic variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

2.12 Choosing mnemonic variable names

As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables. In the beginning, this choice can be confusing both when you read a program and when you write your own programs. For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

```
a = 35.0
b = 12.50
c = a * b
print c

hours = 35.0
rate = 12.50
pay = hours * rate
print pay

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print x1q3p9afd
```

The Python interpreter sees all three of these programs as *exactly the same* but humans see and understand these programs quite differently. Humans will most quickly understand the **intent** of the second program because the programmer has chosen variable names that reflect the intent of the programmer regarding what data will be stored in each variable.

We call these wisely-chosen variable names "menmonic variable names". The word *menmonic*[4] means "memory aid". We choose mnemonic variable names to help us remember why we created the variable in the first place.

While this all sounds great, and it is a very good idea to use mnemonic variable names, mnemonic variable names can get in the way of a beginning programmer's ability to parse and understand code. This is because beginning programmers have not yet memorized the reserved words (there is only

[4]See `http://en.wikipedia.org/wiki/Mnemonic` for an extended description of the word "mnemonic".

31 of them) and sometimes variables which have names that are too descriptive start to look like part of the language and not just well-chosen variable names.

Take a quick look at the following Python sample code which loops through some data. We will cover loops soon, but for now try to just puzzle through what this means:

```
for word in words:
    print word
```

What is happening here? Which of the tokens (for, word, in, etc.) are reserved words and which are just variable names? Does Python understand at a fundamental level the notion of words? Beginning programmers have trouble separating what parts of the code *must* be the same as this example and what parts of the code are simply choices made by the programmer.

The following code is equivalent to the above code:

```
for slice in pizza:
    print slice
```

It is easier for the beginning programmer to look at this code and know which parts are reserved words defined by Python and which parts are simply variable names chosen by the programmer. It is pretty clear that Python has no fundamental understanding of pizza and slices and the fact that a pizza consists of a set of one or more slices.

But if our program is truly about reading data and looking for words in the data, `pizza` and `slice` are very un-mnemonic variable names choosing them as variable names distracts from the meaning of the program.

After a pretty short period of time, you will know the most common reserved words and you will start to see the reserved words jumping out at you:

for `word` **in** `words`**:**
    print `word`

The parts of the code that are defined by Python (`for`, `in`, `print`, and `:`) are in bold and the programmer chosen variables (`word` and `words`) are not in bold. Many text editors are aware of Python syntax and will color reserved words differently to give you clues to keep your variables and reserved words separate. After a while you will begin to read Python and quickly determine what is a variable and what is a reserved word.

2.13 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.

2.14 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.

mnemonic: A memory aid. We often give variables mnemonic names to help us remember what is stored in the variable.

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.

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

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.

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.

2.15 Exercises

Exercise 2.3 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).

1. `width/2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`
5. `delimiter * 5`

Use the Python interpreter to check your answers.

Exercise 2.4 Practice using the Python interpreter as a calculator:

1. 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!
2. 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?
3. 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?

Chapter 3

Conditional execution

3.1 Boolean expressions

A **boolean expression** is an expression 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 **comparison operators**; the others are:

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

Although these operations are probably familiar to you, the 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 comparison operator. There is no such thing as `=<` or `=>`.

3.2 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 might be confusing. You might want to avoid it (unless you know what you are doing).

3.3 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 function definitions: a header followed by an indented block. Statements like this are called **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!
```

3.4 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.

3.5 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.

```
if choice == 'a':
    print "Bad guess"
elif choice == 'b':
    print "Good guess"
elif choice == 'c':
    print "Close, but not correct"
```

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.

3.6 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.'
```

3.7 Catching exceptions

Earlier we saw a code segment where we used the `raw_input` and `int` functions to read and parse an integer number entered by the user. We also saw how treacherous doing this could be:

```
>>> 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()
>>>
```

When we are executing these statements in the Python interpreter, we get a new prompt from the interpreter, think “oops” and move on to our next statement.

However if this code is placed in a Python script and this error occurs, your script immediately stops in its tracks with a traceback. It does not execute the following statement.

Here is a sample program to convert a Fahrenheit temparature to a Celius temparature:

```
inp = raw_input("Enter Fahrenheit Temparature:")
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print cel
```

If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
python fahren.py
Enter Fahrenheit Temparature:72
22.2222222222

python fahren.py
Enter Fahrenheit Temparature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: invalid literal for float(): fred
```

There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called “try / except”. The idea of `try` and `except` is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) is ignored if there is no error.

You can think of the `try` and `except` feature in Python as an “insurance policy” on a sequence of statements.

We can rewrite our temparature converter as follows:

```
inp = raw_input("Enter Fahrenheit Temparature:")
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print cel
except:
    print "Please enter a number"
```

Python starts by executing the sequence of statements in the `try` block. If all goes well, it skips the `except` block and proceeds. If an exception occurs in the `try` block, Python jumps out of the `try` block and executes the sequence of statements in the `except` block.

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

3.8 Short circuit evaluation of logical expressions

When Python is processing a logical expression such as `x >= 2 and (x/y) > 2`, it evaluates the expression from left-to-right. Because of the definition of `and`, if `x` is less than 2, the expression `x >= 2` is `False` and so the whole expression is `False` regardless of whether `(x/y) > 2` evaluates to `True` or `False`.

When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaulation of a logical expression stops because the overall value is already known, it is called **short-circuiting** the evaluation.

While this may seem like a fine point, the short curcuit behavior leads to a clever technique called the **guardian pattern**. Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

The third calculation failed because Python was evaluating `(x/y)` and `y` was zero - which causes a run-time error. But the second example did *not* fail because the first part of the expression `x >= 2` evaluated to `False` so the `(x/y)` was not ever executed due to the **short circuit** rule and there was no error.

We can construct the logical expression to strategically place a **guard** evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

In the first logical expression, `x >= 2` is `False` so the evaluation stops at the `and`. In the second logical expression `x >= 2` is `True` but `y != 0` is `False` so we never reach `(x/y)`.

In the third logical expression, the `y != 0` is *after* the `(x/y)` calculation so the expression fails with an error.

In the second expression, we say that `y != 0` acts as a **guard** to insure that we only execute `(x/y)` if `y` is non-zero.

3.9 Debugging

The traceback Python displays when an error occurs contains a lot of information, but it can be overwhelming, especially when there are many frames on the stack. 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 $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$. 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[1]:

```
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.

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

3.10 Glossary

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

comparison 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.

body: The sequence of statements within a compound statement.

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.

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

short circuit: When Python is part-way through evaluating a logical expression and stops the evaluation because Python knows the final value for the expression without needing to evaluate the rest of the expression.

guardian pattern: Where we construct a logical expression with additional comparisons to take advantage of the short circuit behavior.

[1] In Python 3.0, you no longer get an error message; the division operator performs floating-point division even with integer operands.

3.11 Exercises

Exercise 3.1 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.

1. 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 program should print, "Holy smokes, Fermat was wrong!" Otherwise the program should print, "No, that doesn't work."

2. 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.

Exercise 3.2 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[2]."

1. Write a function named `is_triangle` that takes three integers as arguments, and that prints either "Yes" or "No," depending on whether you can or cannot form a triangle from sticks with the given lengths.

2. 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.

[2] If the sum of two lengths equals the third, they form what is called a "degenerate" triangle.

Chapter 4

Functions

4.1 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**.

4.2 Built-in functions

Python provides a number of important built-in functions that we can use without needing to provide the function definition. In a sense, the creators or Python wrote a set of functions to solve common problems and included them in Python for us to use.

The `max` and `min` functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

The `max` function tells us the "largest character" in the string (which turns out to be the letter "w") and the `min` function shows us the smallest character which turns out to be a space.

Another very common built-in function is the `len` function which tells us how many items are in its argument:

```
>>> len('Hello world')
11
>>>
```

These functions are not limited to looking at strings, they can operate on any set of values as we will see in later chapters.

4.3 Type conversion functions

Python also 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'
```

4.4 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' from '/usr/lib/python2.5/lib-dynload/math.so'>
```

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
```

4.5 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 that execute when the function is called. Once we define a function, we can reuse the function over and over throughout our program.

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 (see Section 4.11). The body can contain any number of statements.

The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

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 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.

4.6 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.

Exercise 4.1 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.

Exercise 4.2 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?

4.7 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 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. Sometimes it makes more sense if you follow the flow of execution.

4.8 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 everybody `bruce`.

4.9 Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them **fruitful functions**. Other functions, like `print_twice`, perform an action but don't return a value. They are called **void functions**.

When you call a fruitful function, 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 fruitful function all by itself, the return value is lost forever!

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

This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called `None`.

```
>>> 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'>
```

To return a result from a function, we use the `return` statement in our function. For example, we could make a very simple function called `addtwo` that adds two numbers together and return a result.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print x
```

When this script executes, the `print` statement will print out "8" because the `addtwo` function was called with 3 and 5 as arguments. Within the function the parameters `a` and `b` were 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named `added` and used the `return` statement to send the computed value back to the calling code as the function result which was assigned to the variable `x` and printed out.

4.10 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, understand 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 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.

Throughout the rest of the book, we often will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as “find the smallest value in a list of values”. Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named `min` which takes a list of values as its argument and returns the smallest value in the list.

4.11 Debugging

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don’t.

Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you.

Also, don’t forget to save your program before you run it. Some development environments do this automatically, but some don’t. In that case the program you are looking at in the text editor is not the same as the program you are running.

Debugging can take a long time if you keep running the same, incorrect, program over and over!

Make sure that the code you are looking at is the code you are running. If you’re not sure, put something like `print 'hello'` at the beginning of the program and run it again. If you don’t see `hello`, you’re not running the right program!

4.12 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.

header: The first line of a function definition.

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.

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.

fruitful function: A function that returns a value.

void function: A function that doesn't return a value.

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 data and code 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.

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

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

4.13 Exercises

Exercise 4.3 Python provides a built-in function called `len` that returns the length of a string, so the value of `len('allen')` is 5.

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 70 of the display.

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

Exercise 4.4 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)
```

1. Type this example into a script and test it.
2. 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.
3. Write a more general version of `print_spam`, called `print_twice`, that takes a string as a parameter and prints it twice.
4. Use the modified version of `do_twice` to call `print_twice` twice, passing `'spam'` as an argument.
5. 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.

You can see my solution at `thinkpython.com/code/do_four.py`.

Exercise 4.5 This exercise[1] can be done using only the statements and other features we have learned so far.

1. Write a function that draws a grid like the following:

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

   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.

2. Use the previous function to draw a similar grid with four rows and four columns.

You can see my solution at `thinkpython.com/code/grid.py`.

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

Chapter 5

Iteration

5.1 Updating variables

One of the most common forms of multiple 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**.

5.2 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. Because iteration is so common, Python provides several language features to make it easier.

One form of iteration in Python is the `while` statement. Here is a simple program that counts down from five and then says "Blastoff!".

```
n = 5
while n > 0:
    print n
    n = n-1
print 'Blastoff!'
```

You can almost read the `while` statement as if it were English. It means, "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, exit the while statement and 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.

This type of flow is called a **loop** because the third step loops back around to the top. Each time we execute the body of the loop, we call it an **iteration**. For the above loop, we would say, "It had five iterations" which means that the body of of the loop was executed five times.

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. We call the variable that changes each time the loop executes and controls when the loop finishes the **iteration variable**. Id there is no iteration variable, the loop will repeat forever, resulting in an **infinite loop**.

5.3 Infinite loops

An endless source of amusement for programmers is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop because there is no **iteration variable** telling you how many times to execute the 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. Other times a loop is obviously infinite because it has no iteration variable at all.

In other cases, it is not so easy to tell. The code below defines a function that takes an positive number as its parameter and computes a different kind of sequence. Remember that the percent sign is the **modulo** operator which gives us the remainder if a division were performed.

```
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.

```
>>> 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
...
>>> sequence(3)
3 10 5 16 8 4 2
>>> sequence(16)
16 8 4 2
>>> sequence(50)
50 25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2
```

You can try this sequence with a variety of integer or floating point numbers as the argument. Since the main loop repeatedly divides a number by two, an argument in the billions converges to one in relatively few steps. It is more fun to try floating point arguments such as 12.45 as it takes more iterations before the sequence converges to one.

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!

5.4 “Infinite loops” and break

Sometimes you don’t know it’s time to end a loop until you get half way through the body. In that case you can write an infinite loop on purpose and then use the break statement to jump out of the loop.

This loop is obviously an **infinite loop** because the logical expression on the while statement is simply the logical constant True:

```
n = 10
while True:
    print n,
    n = n - 1
print 'Done!'
```

If you make the mistake and run this code, you will learn quickly how to stop a runaway Python process on your system or find where the power-off button is on your computer. This program will run forever or until your battery runs out because the logical expression at the top of the loop is always true by virtue of the fact that the expression is the constant value True.

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using break when we have reached the exit condition.

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

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 repeatedly 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:

```
> hello there
hello there
> finished
finished
> done
Done!
```

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.").

5.5 Finishing iterations with `continue`

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the `continue` statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types "done", but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line
print 'Done!'
```

Here is a sample run of this new program with `continue` added.

```
> hello there
hello there
> # don't print this
> print this!
print this!
```

```
> done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the `continue` is executed, it ends the current iteration and jumps back to the `while` statement to start the next iteration, thus skipping the `print` statement.

5.6 Definite loops using `for`

Sometimes we want to loop through a **set** of things such as a list of words, the lines in a file or a list of numbers. When we have some know list of things to loop through, we can construct a *definite* loop using a `for` statement. We call the `while` statement an *indefinite* loop because it simply loops until some condition becomes `False` whereas the `for` loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax of a `for` loop is similar to the `while` loop in that there is a `for` statement and a loop body:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print 'Happy New Year:', friend
print "Done!"
```

Translating this `for` loop to English is not a direct as the `while`, but if you think of friends as a **set**, it goes like this: "Run the statements in the body of the for loop once for each friend *in* the set named friends.".

In Python terms, the variable `friends` is a list[2] of three strings and the `for` loop goes through the list and executes the body once for each of the three strings in the list resulting in this output:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Looking at the `for` loop, **for** and **in** are reserved Python keywords, and `friend` and `friends` are variables.

```
for friend in friends:
    print 'Happy New Year', friend
```

In particular, `friend` is the **iteration variable** for the for loop. The variable `friend` changes for each iteration of the loop and controls when the `for` loop completes. The **iteration variable** steps successively through the three strings stored in the `friends` variable.

5.7 Loop patterns

Often we use a for or while loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.

These loops are generally constructed by:

[2]We will examine lists in more detail in a later chapter

- Create one or more variables and initialize them before the loop starts.
- Look at each of the items in the loop and perform some computation on each item, possibly changing the variables in the body of the loop
- At the end of the loop, the variables contain the information we are looking for

We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

5.7.1 Counting and summing loops

For example, to count the number of items in a list, we would write the following `for` loop:

```
count = 0
for iterval in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print "Count: ", count
```

We set the variable `count` to zero before the loop starts, then we write a `for` loop to run through the list of numbers. Our **iteration** variable is named `itervar` and while we do not use `itervar` in the loop, it does control the loop and cause the loop body to be executed once for each of the values in the list.

In the body of the loop, we add one to the current value of `count` for each of the values in the list. While the loop is executing, the value of `count` is the number of values we have seen "so far".

Once the loop completes, the value of `count` is the total number of items. The total number "falls in our lap" at the end of the loop. We construct the loop so that we have what we want when the loop finishes.

Another similar loop that computes the total of a set of numbers is as follows:

```
total = 0
for iterval in [3, 41, 12, 9, 74, 15]:
    total = total + iterval
print "Total: ", total
```

In this loop we *do* use the **iteration variable**. Instead of simply adding one to the `count` as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running total during each loop iteration. If you think about the variable `total`, it contains the "running total of the values so far". So before the loop starts `total` is zero because we have not yet seen any values, during the loop `total` is the running total, and at the end of the loop `total` is the overal total of all the values in the list.

Neither the counting loop nor the summing loop are particularly useful in practice because there are built-in functions `len()` and `sum()` that compute the number of items in a list and the total of the items in the list respectively.

5.7.2 Maximum and minimum loops

To find the largest value in a list or sequence, we construct the following loop:

```
largest = None
print 'Before:', largest
for iterval in [3, 41, 12, 9, 74, 15]:
    if largest == None or largest < iterval:
        largest = iterval
    print 'Loop:', iterval, largest
print 'Largest:', largest
```

When the program executes, the output is as follows:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

The variable `largest` is best thought of as the "largest value we have seen so far". Before the loop, we set `largest` to the contstant `None`. `None` is a special constant value which we can store in a variable to mark the variable as "empty".

Before the loop starts, the largest value we have seen so far is `None` since we have not yet seen any values. While the loop is executing, if `largest` is `None` then we take the first value we see as the largest so far. You can see in the first iteration when the value of `iterval` is three, since `largest` is `None`, we immediately set `largest` to be three.

After the first iteration, `largest` is no longer `None`, so the second part of the compound logical expression that checks `largest < iterval` triggers only when we see a value that is larger than the "largest so far". When we see a new "even larger" value we take that new value for `largest`. You can see in the program output that `largest` progresses from 3 to 41 to 74.

At the end of the loop, we have scanned all of the values and the variable `largest` now does contain the largest value in the list.

To compute the smallest number, the code is very similar with one small change:

```
smallest = None
print 'Before:', smallest
for iterval in [3, 41, 12, 9, 74, 15]:
    if smallest == None or iterval < smallest:
        smallest = iterval
    print 'Loop:', iterval, smallest
print 'Smallest:', smallest
```

Again, `smallest` is the "smallest so far" before, during, and after the loop executes. When the loop has completed, `smallest` contains the minimum value in the list.

Again as in counting and summing, the built-in functions `max()` and `min()` make writing these exact loops unnecessary.

The following is a simple version of the Python built-in `min()` function:

```
def min(values):
    smallest = None
    for value in values:
        if smallest == None or value < smallest:
            smallest = iterval
    return smallest
```

In the function version of the smallest code, we removed all of the `print` statements so as to be equivalent to the `min` function which is already built-in to Python.

5.8 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, the problem must be 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 halve the number of lines you have to search. After six steps (which is much less 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.

5.9 Glossary

multiple assignment: 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.

initialize: 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.

5.10 Exercises

Exercise 5.1 Need some exercises here.

Chapter 6

Strings

6.1 A string is a sequence

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

```
>>> fruit = 'banana'
>>> 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
a
```

For most people, the first letter of `'banana'` is `b`, not `a`. But in Python, 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
b
```

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

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
```

6.2 `len`

`len` is a built-in function that returns the number of characters in a string:

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

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

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

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

```
>>> last = fruit[length-1]
>>> print last
a
```

Alternatively, 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.

6.3 Traversal with a `for` loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with a `while` loop:

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

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

Exercise 6.1 Write a function that takes a string as an argument and displays the letters backward, one per line.

Another way to write a traversal is with a `for` loop:

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

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

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.

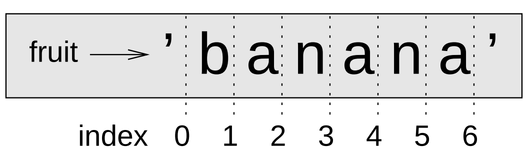
Exercise 6.2 Modify the program to fix this error.

6.4 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[6:13]
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.

Exercise 6.3 Given that `fruit` is a string, what does `fruit[:]` mean?

6.5 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.

6.6 Searching

What does the following function do?

```
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 loop exits normally at the bottom and returns `-1`.

This pattern of computation—traversing a sequence and returning when we find what we are looking for—is a called a **search**.

Exercise 6.4 Modify `find` so that it has a third parameter, the index in `word` where it should start looking.

6.7 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.

Exercise 6.5 Encapsulate this code in a function named `count`, and generalize it so that it accepts the string and the letter as arguments.

Exercise 6.6 Rewrite this function so that instead of traversing the string, it uses the three-parameter version of `find` from the previous section.

6.8 The `in` operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

For example, the following function prints all the letters from `word1` that also appear in `word2`:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print letter
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, "for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter."

Here's what you get if you compare apples and oranges:

```
>>> in_both('apples', 'oranges')
a
e
s
```

6.9 String comparison

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

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

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

```
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
else:
    print '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, so:

```
Your word, 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.

6.10 `string` methods

Strings are an example of Python **objects**. An object contains both data (the actual string itself) as well as **methods** which are effectively functions which are built into the object and available to any **instance** of the object.

Python has a function called `dir` that lists the methods available for an object. The `type` function shows the type of an object and the `dir` function shows the available methods.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
```

```
'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

    Return a copy of the string S with only its first character
    capitalized.
>>>
```

While the `dir` function lists the methods, and you can use `help` to get some simple documentation on a method, a better source of documentation for string methods would be `docs.python.org/library/string.html`.

Calling a **method** is similar to calling a function—it takes arguments and returns a value—but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter.

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
```

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the `strip` method:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

Some methods such as **startswith** return boolean values.

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

You will note that `startswith` requres case to match so sometimes we take a line ane map it all to lower case before we do any checking using the `lower` method.

```
>>> line = 'Please have a nice day'
>>> line.startswith('p')
False
>>> line.lower()
'please have a nice day'
>>> line.lower().startswith('p')
True
```

In the last example, then method `lower` is called and then we use `startswith` check to see if the resulting lower case string starts with the letter “p”. As long as we are careful with the order, we can make multiple method calls in a single expression.

Exercise 6.7 There is a string method called `count` that is similar to the function in the previous exercise. Read the documentation of this method at `docs.python.org/library/string.html` and write an invocation that counts the number of `a`s in `'banana'`.

6.11 Format operator

The **format operator**, `%` allows us to construct strings, replacing parts of the strings with the data stored in variables. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence `'%d'` means that the second operand should be formatted as an integer (`d` stands for “decimal”):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string `'42'`, which is not to be confused with the integer value `42`.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses `'%d'` to format an integer, `'%g'` to format a floating-point number (don't ask why), and `'%s'` to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use. You can read more about it at `docs.python.org/lib/typesseq-strings.html`.

6.12 Debugging

A skill that you should cultivate as you program is always asking yourself, "What could go wrong here?" or alternatively, "What crazy thing might our user do to crash our (seemingly) perfect program?".

For example, look at the program which we used to demonstrate the `while` loop in the chapter on iteration:

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line

print 'Done!'
```

Look what happens when the user enters an empty line of input:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
```

```
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#' :
```

The code works fine until it is presented an empty line. Then there is no zeroth character so we get a traceback. There are two solutions to this to make line three ”safe” even if the line is empty.

One possibility is to simply use the `startswith` method which returns `False` if the string is empty.

```
    if line.startswith('#') :
```

Another way to safely write the if statment using the **guardian** pattern and make sure the second logical expression is evaluated only where there is at least one character in the string.:

```
    if len(line) > 0 and line[0] == '#' :
```

Another common source of problems is when you hand-construct index values to move through a sequence. It can be quite 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. This is another example of the guardian pattern.

`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:

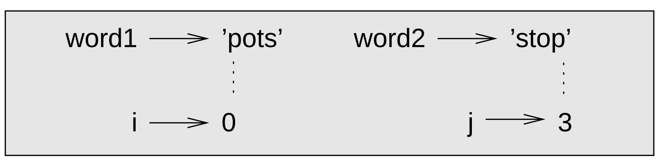
```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

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`.

Exercise 6.8 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.

6.13 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.

6.14 Exercises

Exercise 6.9 A string slice can take a third index that specifies the "step size;" that is, the number of spaces between successive characters. A step size of 2 means every other character; 3 means every third, etc.

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

A step size of -1 goes through the word backwards, so the slice `[::-1]` generates a reversed string.

Use this idiom to write a one-line version of `is_palindrome` from Exercise 8.5.

Exercise 6.10 Read the documentation of the string methods at `docs.python.org/lib/string-methods.html`. You might want to experiment with some of them to make sure you understand how they work. `strip` and `replace` are particularly useful.

The documentation uses a syntax that might be confusing. For example, in `find(sub[, start[, end]])`, the brackets indicate optional arguments. So `sub` is required, but `start` is optional, and if you include `start`, then `end` is optional.

Exercise 6.11 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
```

Exercise 6.12 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”.

You might want to use the built-in functions `ord`, which converts a character to a numeric code, and `chr`, which converts numeric codes to characters.

Potentially offensive jokes on the Internet are sometimes encoded in ROT13. If you are not easily offended, find and decode some of them.

[1]See `wikipedia.org/wiki/ROT13`

Chapter 7

Files

7.1 Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapters we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, `pickle`, that makes it easy to store program data.

7.2 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 in Section **??**.

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)
```

When you are done writing, you have to close the file.

```
>>> fout.close()
```

7.3 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('/home')
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.

7.4 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 `wikipedia.org/wiki/Newline`. Or, of course, you could write one yourself.

7.5 Glossary

persistent: Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.

format operator: An operator, `%`, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format string: A string, used with the format operator, that contains format sequences.

format sequence: A sequence of characters in a format string, like `%d`, that specifies how a value 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.

database: A file whose contents are organized like a dictionary with keys that correspond to values.

7.6 Exercises

Exercise 7.1 The Internet Movie Database (IMDb) is an online collection of information about movies. Their database is available in plain text format, so it is reasonably easy to read from Python. For this exercise, the files you need are `actors.list.gz` and `actresses.list.gz`; you can download them from `www.imdb.com/interfaces#plain`.

I have written a program that parses these files and splits them into actor names, movie titles, etc. You can download it from `thinkpython.com/code/imdb.py`.

If you run `imdb.py` as a script, it reads `actors.list.gz` and prints one actor-movie pair per line. Or, if you `import imdb` you can use the function `process_file` to, well, process the file. The arguments are a filename, a function object and an optional number of lines to process. Here is an example:

```
import imdb

def print_info(actor, date, title, role):
    print actor, date, title, role

imdb.process_file('actors.list.gz', print_info)
```

When you call `process_file`, it opens `filename`, reads the contents, and calls `print_info` once for each line in the file. `print_info` takes an actor, date, movie title and role as arguments and prints them.

1. Write a program that reads `actors.list.gz` and `actresses.list.gz` and uses `shelve` to build a database that maps from each actor to a list of his or her films.

2. Two actors are "costars" if they have been in at least one movie together. Process the database you built in the previous step and build a second database that maps from each actor to a list of his or her costars.

3. Write a program that can play the "Six Degrees of Kevin Bacon," which you can read about at `wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon`. This problem is challenging because it requires you to find the shortest path in a graph. You can read about shortest path algorithms at `wikipedia.org/wiki/Shortest_path_problem`.

Chapter 8

More on functions

8.1 Return values

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

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

All of the functions we have written so far are void; they print something or move turtles around, but their return value is `None`.

In this chapter, we are (finally) going to write fruitful functions. The first example is `area`, which returns the area of a circle with the given radius:

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

We have seen the `return` statement before, but in a fruitful function the `return` statement includes an expression. This 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 fruitful function, it is a good idea to ensure that every possible path through the program hits a `return` statement. For example:

```
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.

Exercise 8.1 Write a `compare` function that returns `1` if `x > y`, `0` if `x == y`, and `-1` if `x < y`.

8.2 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`.

8.3 Global variables

In the previous example, `known` is created outside the function, so it belongs to the special frame called `__main__`. Variables in `__main__` are sometimes called **global** because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next.

It is common to use global variables for **flags**; that is, boolean variables that indicate ("flag") whether a condition is true. For example, some programs use a flag named `verbose` to control the level of detail in the output:

```
verbose = True

def example1():
    if verbose:
        print 'Running example1'
```

If you try to reassign a global variable, you might be surprised. The following example is supposed to keep track of whether the function has been called:

```
been_called = False

def example2():
    been_called = True         # WRONG
```

But if you run it you will see that the value of `been_called` doesn't change. The problem is that `example2` creates a new local variable named `been_called`. The local variable goes away when the function ends, and has no effect on the global variable.

To reassign a global variable inside a function you have to **declare** the global variable before you use it:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

The `global` statement tells the interpreter something like, "In this function, when I say `been_called`, I mean the global variable; don't create a local one."

Here's an example that tries to update a global variable:

```
count = 0

def example3():
    count = count + 1          # WRONG
```

If you run it you get:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python assumes that `count` is local, which means that you are reading it before writing it. The solution, again, is to declare `count` global.

```
def example3():
    global count
    count += 1
```

If the global value is mutable, you can modify it without declaring it:

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

So you can add, remove and replace elements of a global list or dictionary, but if you want to reassign the variable, you have to declare it:

```
def example5():
    global known
    known = dict()
```

8.4 Incremental development

As you write larger functions, you might find yourself spending more time debugging.

To deal with 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. 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.

Exercise 8.2 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.

8.5 Composition

As you should expect by now, you can call one function from within another. This ability is called **composition**.

As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. We just wrote a function, `distance`, that does that:

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

The next step is to find the area of a circle with that radius; we just wrote that, too:

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

Encapsulating these steps in a function, we get:

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

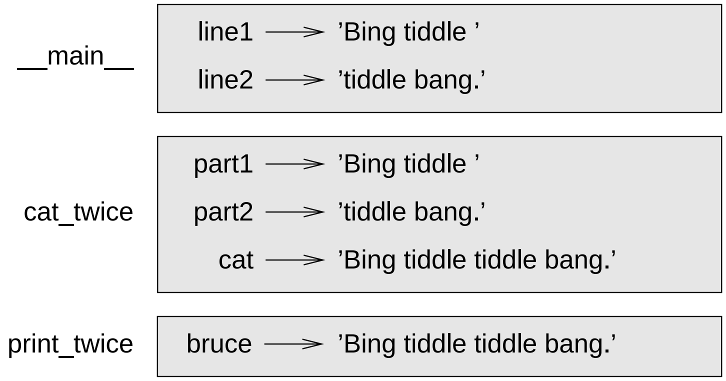
The temporary variables `radius` and `result` are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

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

8.6 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**. 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.

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.

8.7 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.

Exercise 8.3 Write a function `is_between(x, y, z)` that returns `True` if $x \leq y \leq z$ or `False` otherwise.

8.8 Debugging

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; a precondition is violated.
- There is something wrong with the function; a postcondition is violated.
- 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 (as in Section 8.4).

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!).

Adding print statements at the beginning and end of a function can help make the flow of execution more visible.

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.

8.9 Glossary

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.

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

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.

guardian: A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

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.

8.10 Exercises

Exercise 8.4 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)
```

Exercise 8.5 A palindrome is a word that is spelled the same backward and forward, like "noon" and "redivider". Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

The following are functions that take a string argument and return the first, last, and middle letters:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

We'll see how they work in Chapter 6.

1. Type these functions into a file named `palindrome.py` and test them out. What happens if you call `middle` with a string with two letters? One letter? What about the empty string, which is written `''` and contains no letters?

2. Write a function called `is_palindrome` that takes a string argument and returns `True` if it is a palindrome and `False` otherwise. Remember that you can use the built-in function `len` to check the length of a string.

Exercise 8.6 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`.

Exercise 8.7 The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder[1].

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 `wikipedia.org/wiki/Euclidean_algorithm`.

[1] This exercise is based on an example from Abelson and Sussman's *Structure and Interpretation of Computer Programs*.

Chapter 9

Lists

9.1 A list is a sequence

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 list are called **elements** or sometimes **items**.

There are several ways to create a new list; the simplest is to enclose the 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. 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] []
```

9.2 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:

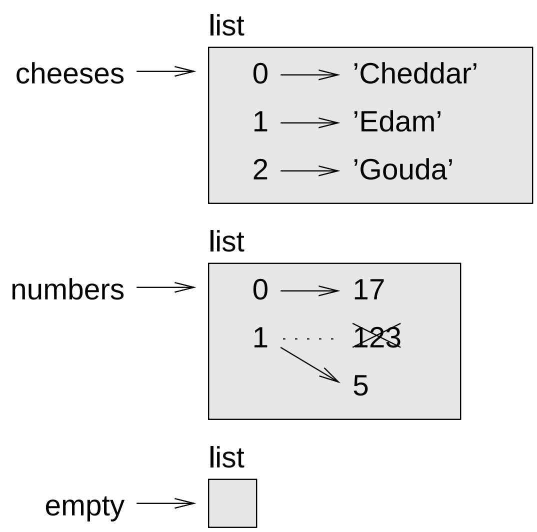
```
>>> print cheeses[0]
Cheddar
```

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 assigned.

```
>>> 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`:



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
```

9.3 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 empty:
    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]]
```

9.4 List operations

The `+` operator concatenates lists:

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

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

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

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

9.5 List slices

The slice operator 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.

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']
```

9.6 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']
```

List methods are all void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

9.7 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. The `+=` operator provides a short way to update a variable:

```
    total += x
```

is equivalent to:

```
    total = total + x
```

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 it as a 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.

Most common list operations can be expressed as a combination of map, filter and reduce. Because these operations are so common, Python provides language features to support them, including the built-in function `map` and an operator called a “list comprehension.”

Exercise 9.1 Write a function that takes a list of numbers and returns the cumulative sum; that is, a new list where the *i*th element is the sum of the first $i+1$ elements from the original list. For example, the cumulative sum of `[1, 2, 3]` is `[1, 3, 6]`.

9.8 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.

9.9 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 I 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:

```
>>> 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'
```

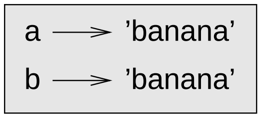
In this case the delimiter is a space character, so `join` puts a space between words. To concatenate strings without spaces, you can use the empty string, `''`, as a delimiter.

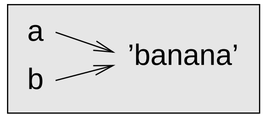
9.10 Objects and values

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:





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.

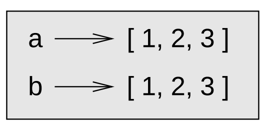
```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

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:



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.

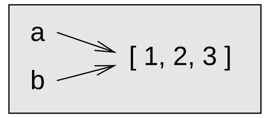
Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value. If you execute `a = [1,2,3]`, `a` refers to a list object whose value is a particular sequence of elements. If another list has the same elements, we would say it has the same value.

9.11 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.

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.

9.12 List arguments

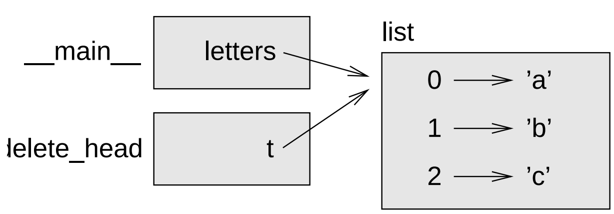
When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
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:



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']
```

Exercise 9.2 Write a function called `chop` that takes a list and modifies it, removing the first and last elements, and returns `None`.

Then write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements.

9.13 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 `docs.python.org/lib/typesseq.html`. The methods and operators that only apply to mutable sequences are documented at `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. But don't forget that 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!

9.14 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.

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.

9.15 Exercises

Exercise 9.3 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 comparison operators `<`, `>`, etc.

For example, `is_sorted([1,2,2])` should return `True` and `is_sorted(['b','a'])` should return `False`.

Exercise 9.4 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.

Exercise 9.5 The (so-called) Birthday Paradox:

1. 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.

2. 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 `wikipedia.org/wiki/Birthday_paradox`, and you can see my solution at `thinkpython.com/code/birthday.py`.

Exercise 9.6 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.

Exercise 9.7 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?

You can see my solution at `thinkpython.com/code/wordlist.py`.

Exercise 9.8 To check whether a word is in the word list, you could use the `in` operator, but it would be slow because it searches through the words in order.

Because the words are in alphabetical order, we can speed things up with a bisection 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 113,809 words, it will take about 17 steps to find the word or conclude that it's not there.

Write a function called `bisect` that takes a sorted list and a target value and returns the index of the value in the list, if it's there, or `None` if it's not.

Or you could read the documentation of the `bisect` module and use that!

Exercise 9.9 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.

Exercise 9.10 Two words "interlock" if taking alternating letters from each forms a new word[1]. For example, "shoe" and "cold" interlock to form "schooled."

1. Write a program that finds all pairs of words that interlock. Hint: don't enumerate all pairs!
2. 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?

[1]This exercise is inspired by an example at `puzzlers.org`.

Chapter 10

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**.

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 Section 6.6. 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 `wikipedia.org/wiki/Hash_table`.

Exercise 10.1 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.

If you did Exercise 9.8, you can compare the speed of this implementation with the list `in` operator and the bisection search.

10.1 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.

Exercise 10.2 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.

10.2 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.

Exercise 10.3 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.

10.3 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.

Exercise 10.4 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.

10.4 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]
```

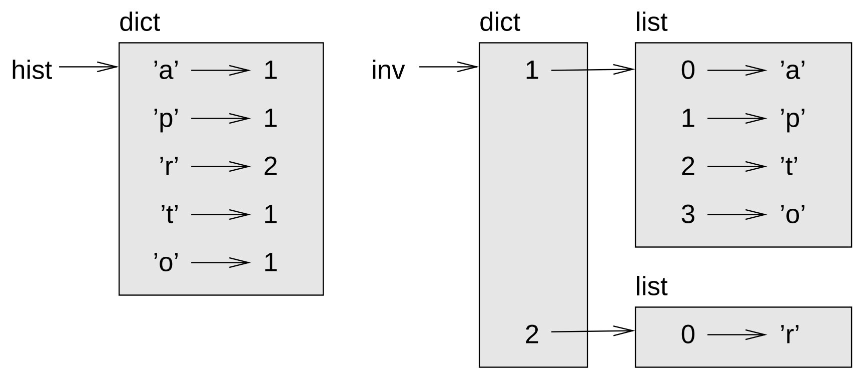
```
        else:
            inv[val].append(key)
    return inv
```

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:



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.

Exercise 10.5 Read the documentation of the dictionary method `setdefault` and use it to write a more concise version of `invert_dict`.

10.5 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."

Another kind of check compares the results of two different computations to see if they are consistent. This is called a "consistency check."

Pretty print the output: Formatting debugging output can make it easier to spot an error. We saw an example in Section 8.8. The `pprint` module provides a `pprint` function that displays built-in types in a more human-readable format.

Again, time you spend building scaffolding can reduce the time you spend debugging.

10.6 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.

global variable: A variable defined outside a function. Global variables can be accessed from any function.

flag: A boolean variable used to indicate whether a condition is true.

declaration: A statement like `global` that tells the interpreter something about a variable.

10.7 Exercises

Exercise 10.6 If you did Exercise 9.5, you already have a function named `has_duplicates` that takes a list as a parameter and returns `True` if there is any object that appears more than once in the list.

Use a dictionary to write a faster, simpler version of `has_duplicates`.

Exercise 10.7 Two words are "rotate pairs" if you can rotate one of them and get the other (see `rotate_word` in Exercise 6.12).

Write a program that reads a wordlist and finds all the rotate pairs.

Exercise 10.8 Here's another Puzzler from *Car Talk*[1]:

> 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?

[1] `www.cartalk.com/content/puzzler/transcripts/200717`

> 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.' 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?

You can use the dictionary from Exercise 10.1 to check whether a string is in the word list.

To check whether two words are homophones, you can use the CMU Pronouncing Dictionary. You can download it from `www.speech.cs.cmu.edu/cgi-bin/cmudict` or from `thinkpython.com/code/c06d` and you can also download `thinkpython.com/code/pronounce.py`, which provides a function named `read_dictionary` that reads the pronouncing dictionary and returns a Python dictionary that maps from each word to a string that describes its primary pronunciation.

Write a program that lists all the words that solve the Puzzler. You can see my solution at `thinkpython.com/code/homophone.py`.

Chapter 11

Tuples

11.1 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 the final comma:

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

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

```
>>> 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 indexes 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')
```

11.2 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
```

11.3 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.

11.4 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')
```

You can combine the gather operator with required and positional arguments:

```
def pointless(required, optional=0, *args):
    print required, optional, args
```

Run this function with 1, 2, 3 and 4 or more arguments and make sure you understand what it does.

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)
```

Exercise 11.1 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.

11.5 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:

[1] In Python 3.0, `zip` returns an iterator of tuples, but for most purposes, an iterator behaves like a list.

```
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]`:

```
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.

11.6 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}
```

[2]This behavior is slightly different in Python 3.0.

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.

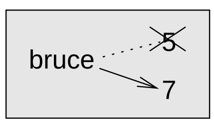
11.7 Multiple assignment

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

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

The output of this program is `5 7`, 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 **multiple assignment** looks like in a state diagram:



With multiple assignment it is especially important to distinguish between an assignment operation and a statement of equality. Because 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 multiple assignment is frequently helpful, you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

11.8 Multiple assignment with dictionaries

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
```

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:

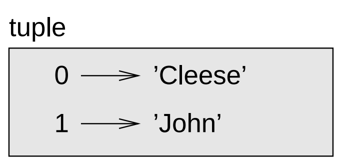
```
directory[last,first] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

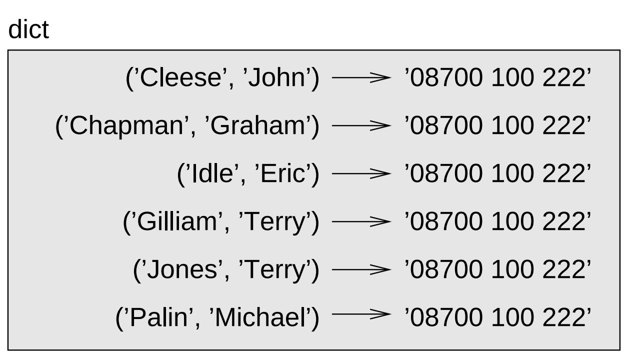
```
for last, first in directory:
    print first, last, directory[last,first]
```

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:



But in a larger diagram you might want to leave out the details. For example, a diagram of the telephone directory might appear:



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.

11.9 Comparing tuples

The comparison 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.

Exercise 11.2 In this example, ties are broken by comparing words, so words with the same length appear in 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.

11.10 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.

11.11 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.

To help debug these kinds of errors, I have written a module called `structshape` that provides a function, also called `structshape`, that takes any kind of data structure as an argument and returns a string that summarizes its shape. You can download it from `thinkpython.com/code/structshape.py`

Here's the result for a simple list:

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> print structshape(t)
list of 3 int
```

A fancier program might write "list of 3 int*s*," but it was easier not to deal with plurals. Here's a list of lists:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> print structshape(t2)
list of 3 list of 2 int
```

If the elements of the list are not the same type, `structshape` groups them, in order, by type:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> print structshape(t3)
list of (3 int, float, 2 str, 2 list of int, int)
```

Here's a list of tuples:

```
>>> s = 'abc'
>>> lt = zip(t, s)
>>> print structshape(lt)
list of 3 tuple of (int, str)
```

And here's a dictionary with 3 items that map integers to strings.

```
>>> d = dict(lt)
>>> print structshape(d)
dict of 3 int->str
```

If you are having trouble keeping track of your data structures, `structshape` can help.

11.12 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.

11.13 Exercises

Exercise 11.3 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 `wikipedia.org/wiki/Letter_frequencies`.

Exercise 11.4 More anagrams!

1. Write a program that reads a word list from a file (see 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?

2. Modify the previous program so that it prints the largest set of anagrams first, followed by the second largest set, and so on.

3. 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.

4. 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.

   You can download a solution from `thinkpython.com/code/anagram_sets.py`.

Exercise 11.5 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:

1. 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.

2. Recursively, a word is reducible if any of its children are reducible. As a base case, you can consider the empty string reducible.

3. The wordlist I provided, `words.txt`, doesn't contain single letter words. So you might want to add "I", "a", and the empty string.

4. To improve the performance of your program, you might want to memoize the words that are known to be reducible.

You can see my solution at `thinkpython.com/code/reducible.py`.

[3] This exercise is inspired by an example at `puzzlers.org`.
[4] `www.cartalk.com/content/puzzler/transcripts/200651`

Chapter 12

Case study: data structure selection

12.1 Word frequency analysis

As usual, you should at least attempt the following exercises before you read my solutions.

Exercise 12.1 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`.

Exercise 12.2 Go to Project Gutenberg (`gutenberg.net`) 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.

Then modify the program to count the total number of words in the book, and the number of times each word is used.

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?

Exercise 12.3 Modify the program from the previous exercise to print the 20 most frequently-used words in the book.

Exercise 12.4 Modify the previous program to read a word list (see Section **??**) 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?

12.2 Random numbers

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 I will simply call "random" from here 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
```

To choose an element from a sequence at random, you can use `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The `random` module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

Exercise 12.5 Write a function named `choose_from_hist` that takes a histogram as defined in Section 10.1 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 $1/3$.

12.3 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
```

12.4 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
```

12.5 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.

12.6 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!

Exercise 12.6 Python provides a data structure called `set` that provides many common set operations. Read the documentation at `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.

12.7 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.

Exercise 12.7 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:

1. Use `keys` to get a list of the words in the book.
2. Build a list that contains the cumulative sum of the word frequencies (see Exercise 9.1). The last item in this list is the total number of words in the book, n.
3. Choose a random number from 1 to n. Use a bisection search (See Exercise 9.8) to find the index where the random number would be inserted in the cumulative sum.
4. 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.

12.8 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.

12.9 Glossary

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.

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.

12.10 Exercises

Exercise 12.8 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[1]. 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:

[1] See `wikipedia.org/wiki/Zipf's_law`

$$\log f = \log c - s \log r$$

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?

Chapter 13

Networked programs

13.1 The Internet

13.2 Making network connections

13.3 Retrieving web pages

13.4 Parsing HTML and scraping the web

13.5 Exercises

Exercise 13.1 The `urllib` module provides methods for manipulating URLs and downloading information from the web. The following example downloads and prints a secret message from `thinkpython.com`:

```
import urllib

conn = urllib.urlopen('http://thinkpython.com/secret.html')
for line in conn.fp:
    print line.strip()
```

Run this code and follow the instructions you see there.

Chapter 14

Using web services

14.1 Serivce Oriented Architeture (SOA)

14.2 Application Programming Intercaces (API)

14.3 eXtensible Markup Language (XML)

14.4 Parsing XML

14.5 Calling web services

Chapter 15

Using databases

15.1 What is a database?

A **database** is a file that is organized for storing data. Most databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module `anydbm` provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.

Opening a database is similar to opening other files:

```
>>> import anydbm
>>> db = anydbm.open('captions.db', 'c')
```

The mode `'c'` means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary. If you create a new item, `anydbm` updates the database file.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, `anydbm` reads the file:

```
>>> print db['cleese.png']
Photo of John Cleese.
```

If you make another assignment to an existing key, `anydbm` replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> print db['cleese.png']
Photo of John Cleese doing a silly walk.
```

Many dictionary methods, like `keys` and `items`, also work with database objects. So does iteration with a `for` statement.

```
for key in db:
     print key
```

As with other files, you should close the database when you are done:

```
>>> db.close()
```

15.2 Pickling

A limitation of anydbm is that the keys and values have to be strings. If you try to use any other type, you get an error.

The pickle module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

pickle.dumps takes an object as a parameter and returns a string representation (dumps is short for "dump string"):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na.'
```

The format isn't obvious to human readers; it is meant to be easy for pickle to interpret. pickle.loads ("load string") reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> print t2
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

You can use pickle to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called shelve.

Exercise 15.1 If you did Exercise 11.4, modify your solution so that it creates a database that maps from each word in the list to a list of words that use the same set of letters.

Write a different program that opens the database and prints the contents in a human-readable format.

15.3 Using the SQLite3 database

15.4 The basics of Structured Query Language (SQL)

15.5 Basic data modelling

15.6 Using JOIN to retrieve data

15.7 Python and SQLite3

Chapter 16

Regular expressions

Chapter 17

Automating common tasks

17.1 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
/home/dinsdale
```

`cwd` stands for “current working directory.” The result in this example is `/home/dinsdale`, which is the home directory of a user named `dinsdale`.

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')
'/home/dinsdale/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 17.1 Modify `walk` so that instead of printing the names of the files, it returns a list of names.

Exercise 17.2 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.

17.2 Pipes

Most operating systems provide a command-line interface, also known as a **shell**. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix, you can change directories with `cd`, display the contents of a directory with `ls`, and launch a web browser by typing (for example) `firefox`.

Any program that you can launch from the shell can also be launched from Python using a **pipe**. A pipe is an object that represents a running process.

For example, the Unix command `ls -l` normally displays the contents of the current directory (in long format). You can launch `ls` with `os.popen`:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

The argument is a string that contains a shell command. The return value is a file pointer that behaves just like an open file. You can read the output from the `ls` process one line at a time with `readline` or get the whole thing at once with `read`:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print stat
None
```

The return value is the final status of the `ls` process; `None` means that it ended normally (with no errors).

A common use of pipes is to read a compressed file incrementally; that is, without uncompressing the whole thing at once. The following function takes the name of a compressed file as a parameter and returns a pipe that uses `gunzip` to decompress the contents:

```
def open_gunzip(filename):
    cmd = 'gunzip -c ' + filename
    fp = os.popen(cmd)
    return fp
```

If you read lines from `fp` one at a time, you never have to store the uncompressed file in memory or on disk.

17.3 Exercises

Exercise 17.3 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.

1. 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.

2. 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 `wikipedia.org/wiki/Md5`. On a Unix system you can use the program `md5sum` and a pipe to compute checksums from Python.

Chapter 18

Visualizing data

18.1 Simple graphics in Python

18.2 Python and spreadsheets

18.3 Using Google visualization

Appendix A

Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

- Syntax errors are produced by 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.
- 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.

A.1 Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where 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...

A.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.

A.2 Runtime errors

Once your program is syntactically correct, Python can compile it and at least start running it. What could possibly go wrong?

A.2.1 My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the “Flow of Execution” section below.

A.2.2 My program hangs.

If a program stops and seems to be doing nothing, it is “hanging.” Often that means that it is caught in an infinite loop. 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. If you think you have an infinite loop and you think you know what loop is causing the problem, add a `print` statement inside the loop as the last statement in the loop 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.

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.

A.2.3 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 `docs.python.org/lib/module-pdb.html`.

A.2.4 I added so many `print` statements I get inundated with output.

One of the problems with using `print` statements for debugging is that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out `print` statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are 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.

A.3 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.

A.3.1 My program doesn't work.

You should ask yourself these questions:

- Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to functions or methods in other 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.

A.3.2 I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
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.

A.3.3 I've got a function or method that doesn't return what I expect.

If you have a `return` statement with a complex expression, you don't have a chance to print the `return` value before returning. Again, you can use a temporary variable. For example, instead of:

```
return 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.

A.3.4 I'm really, really stuck and I need help.

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these 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.

A.3.5 No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you 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.

Appendix B

Contributor List

Contributor List for “Python for Informatics”

TBD

Contributor List for “Think Python”

(Allen B. Downey)

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.

For the detail on the nature of each of the contributions from these individuals, see the “Think Python” text.

Lloyd Hugh Allen, Yvon Boulianne, Fred Bremmer, Jonah Cohen, Michael Conlon, Benoit Girard, Courtney Gleason and Katherine Smith, Lee Harr, James Kaylin, David Kershaw, Eddie Lam, Man-Yong Lee, David Mayo, Chris McAloon, Matthew J. Moelter, Simon Dicon Montford, John Ouzts, Kevin Parks, David Pool, Michael Schmitt, Robin Shaw, Paul Sleigh, Craig T. Snydal, Ian Thomas, Keith Verheyden, Peter Winstanley, Chris Wrobel, Moshe Zadka, Christoph Zwerschke, James Mayer, Hayden McAfee, Angel Arnal, Tauhidul Hoque and Lex Berezhny, Dr. Michele Alzetta, Andy Mitchell, Kalin Harvey, Christopher P. Smith, David Hutchins, Gregor Lingl, Julie Peters, Florin Oprina, D. J. Webre, Ken, Ivo Wever, Curtis Yanko, Ben Logan, Jason Armstrong, Louis Cordier, Brian Cain, Rob Black, Jean-Philippe Rey at Ecole Centrale Paris, Jason Mader at George Washington University made a number Jan Gundtofte-Bruun, Abel David and Alexis Dinno, Charles Thayer, Roger Sperberg, Sam Bull, Andrew Cheung, C. Corey Capel, Alessandra, Wim Champagne, Douglas Wright, Jared Spindor, Lin Peiheng, Ray Hagtvedt, Torsten Hübsch, Inga Petuhhov, Arne Babenhauserheide, Mark E. Casida, Scott Tyler, Gordon Shephard, Andrew Turner, Adam Hobart, Daryl Hammond and Sarah Zimmerman, George Sass, Brian Bingham, Leah Engelbert-Fenton, Joe Funke, Chao-chao Chen, Jeff Paine, Lubos Pintes, Gregg Lind and Abigail Heithoff, Max Hailperin, Chotipat Pornavalai, Stanislaw Antol, Eric Pashman, Miguel Azevedo, Jianhua Liu, Nick King, Martin Zuther, Adam Zimmerman, Ratnakar Tiwari, Anurag Goel, Kelli Kratzer, Mark Griffiths, Roydan Ongie, Patryk Wolowiec, Mark Chonofsky, Russell Coleman, Wei Huang, Karen Barber, Nam Nguyen, Stéphane Morin, and Paul Stoop.