Think Python

How to Think Like a Computer Scientist

Version 1.1.20

Think Python

How to Think Like a Computer Scientist

Version 1.1.20

Allen Downey

Green Tea Press

Needham, Massachusetts

Printing history:

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

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

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

Green Tea Press
9 Washburn Ave
Needham MA 02492

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 this book is available from `http://www.thinkpython.com`

Preface

The strange history of this book

In January 1999 I was preparing to teach an introductory programming class in Java. I had taught it three times and I was getting frustrated. The failure rate in the class was too high and, even for students who succeeded, the overall level of achievement was too low.

One of the problems I saw was the books. They were too big, with too much unnecessary detail about Java, and not enough high-level guidance about how to program. And they all suffered from the trap door effect: they would start out easy, proceed gradually, and then somewhere around Chapter 5 the bottom would fall out. The students would get too much new material, too fast, and I would spend the rest of the semester picking up the pieces.

Two weeks before the first day of classes, I decided to write my own book. My goals were:

- Keep it short. It is better for students to read 10 pages than not read 50 pages. • Be careful

  with vocabulary. I tried to minimize the jargon and define each term at first use.

- Build gradually. To avoid trap doors, I took the most difficult topics and split them into a series of small steps.

- Focus on programming, not the programming language. I included the minimum useful subset of Java and left out the rest.

I needed a title, so on a whim I chose *How to Think Like a Computer Scientist*.

My first version was rough, but it worked. Students did the reading, and they understood enough that I could spend class time on the hard topics, the interesting topics and (most important) letting the students practice.

I released the book under the GNU Free Documentation License, which allows users to copy, modify, and distribute the book.

What happened next is the cool part. Jeff Elkner, a high school teacher in Virginia, adopted my book and translated it into Python. He sent me a copy of his translation, and I had the unusual experience of learning Python by reading my own book.

Jeff and I revised the book, incorporated a case study by Chris Meyers, and in 2001 we released *How to Think Like a Computer Scientist: Learning with Python*, also under the GNU Free Doc umentation 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!

The result is this book, now with the less grandiose title *Think Python*. Some of the changes are:

- I added a section about debugging at the end of each chapter. These sections present general techniques for finding and avoiding bugs, and warnings about Python pitfalls.

- I removed the material in the last few chapters about the implementation of lists and trees. I still love those topics, but I thought they were incongruent with the rest of the book.
- I added more exercises, ranging from short tests of understanding to a few substantial projects.
- I added a series of case studies—longer examples with exercises, solutions, and discussion. Some of them are based on Swampy, a suite of Python programs I wrote for use in my classes. Swampy, code examples, and some solutions are available from `thinkpython.com`.
- I expanded the discussion of program development plans and basic design patterns.
- The use of Python is more idiomatic. The book is still about programming, not Python, but now I think the book gets more leverage from the language.

I hope you enjoy working with this book, and that it helps you learn to program and think, at least a little bit, like a computer scientist.

Allen B. Downey
Needham MA

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

Acknowledgements

First and most importantly, I thank Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language.

I also thank Chris Meyers, who contributed several sections to *How to Think Like a Computer Scientist*.

And I thank the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible.

I also thank the editors at Lulu who worked on *How to Think Like a Computer Scientist*.

I thank all the students who worked with earlier versions of this book and all the contributors (listed below) who sent in corrections and suggestions.

And I thank my wife, Lisa, for her work on this book, and Green Tea Press, and everything else, too.

Contributor List

More than 100 sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

If you have a suggestion or correction, please send email to `feedback@thinkpython.com`. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.

• Benoit Girard sent in a correction to a humorous mistake in Section 5.6.

• Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.

• Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.

• James Kaylin is a student using the text. He has submitted numerous corrections.

• David Kershaw fixed the broken `catTwice` function in Section 3.10.

• Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.

• Man-Yong Lee sent in a correction to the example code in Section 2.4.

• David Mayo pointed out that the word "unconsciously" in Chapter 1 needed to be changed to "subcon sciously".

• Chris McAloon sent in several corrections to Sections 3.9 and 3.10.

• Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.

• Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the `increment` function in Chapter 13.

• John Ouzts corrected the definition of "return value" in Chapter 3.

• Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.

• David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.

• Michael Schmitt sent in a correction to the chapter on files and exceptions.

• Robin Shaw pointed out an error in Section 13.1, where the printTime function was used in an example without being defined.

• Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX.

• Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.

• Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.

• Keith Verheyden sent in a correction in Chapter 3.

• Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3. •

Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.

• Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.

• Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between *gleich* and *selbe*.

• James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.

• Hayden McAfee caught a potentially confusing inconsistency between two examples.

• Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.

• Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.

• Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.

• Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.

• Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.

• Christopher P. Smith caught several typos and is helping us prepare to update the book for Python 2.2.

• David Hutchins caught a typo in the Foreword.

• Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German

translation of the book, and he caught a couple of bad errors in Chapter 5.

• Julie Peters caught a typo in the Preface.

• Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice

typo. • D. J. Webre suggested a clarification in Chapter 3.

• Ken found a fistful of errors in Chapters 8, 9 and 11.

• Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.

• Curtis Yanko suggested a clarification in Chapter 2.

• Ben Logan sent in a number of typos and problems with translating the book into HTML.

• Jason Armstrong saw the missing word in Chapter 2.

• Louis Cordier noticed a spot in Chapter 16 where the code didn't match the text.

• Brian Cain suggested several clarifications in Chapters 2 and 3.

• Rob Black sent in a passel of corrections, including some changes for Python 2.2.

• Jean-Philippe Rey at Ecole Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.

• Jason Mader at George Washington University made a number of useful suggestions and corrections.

• Jan Gundtofte-Bruun reminded us that "a error" is an error.

• Abel David and Alexis Dinno reminded us that the plural of "matrix" is "matrices", not "matrixes". This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.

• Charles Thayer encouraged us to get rid of the semi-colons we had put at the ends of some statements and to clean up our use of "argument" and "parameter".

• Roger Sperberg pointed out a twisted piece of logic in Chapter 3.

• Sam Bull pointed out a confusing paragraph in Chapter 2.

• Andrew Cheung pointed out two instances of "use before def."

• C. Corey Capel spotted the missing word in the Third Theorem of Debugging and a typo in Chapter 4.

• Alessandra helped clear up some Turtle confusion.

• Wim Champagne found a brain-o in a dictionary example.

• Douglas Wright pointed out a problem with floor division in `arc`.

• Jared Spindor found some jetsam at the end of a sentence.

• Lin Peiheng sent a number of very helpful suggestions.

• Ray Hagtvedt sent in two errors and a not-quite-error.

• Torsten Hubsch pointed out an inconsistency in Swampy. ¨

• Inga Petuhhov corrected an example in Chapter 14.

• Arne Babenhauserheide sent several helpful corrections.

• Mark E. Casida is is good at spotting repeated words.

• Scott Tyler filled in a that was missing. And then sent in a heap of corrections.

• Gordon Shephard sent in several corrections, all in separate emails.

• Andrew Turner `spot`ted an error in Chapter 8.

• Adam Hobart fixed a problem with floor division in `arc`.

• Daryl Hammond and Sarah Zimmerman pointed out that I served up `math.pi` too early. And Zim spotted a typo.

• George Sass found a bug in a Debugging section.

• Brian Bingham suggested Exercise 11.9.

• Leah Engelbert-Fenton pointed out that I used `tuple` as a variable name, contrary to my own advice. And then found a bunch of typos and a "use before def."

• Joe Funke spotted a typo.

• Chao-chao Chen found an inconsistency in the Fibonacci example.

• Jeff Paine knows the difference between space and spam.

• Lubos Pintes sent in a typo.

• Gregg Lind and Abigail Heithoff suggested Exercise 14.6.

• Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary *Concrete Abstractions*, which you might want to read when you are done with this book.

• Chotipat Pornavalai found an error in an error message.

• Stanislaw Antol sent a list of very helpful suggestions.

• Eric Pashman sent a number of corrections for Chapters 4–11.

• Miguel Azevedo found some typos.

• Jianhua Liu sent in a long list of corrections.

• Nick King found a missing word.

• Martin Zuther sent a long list of suggestions.

• Adam Zimmerman found an inconsistency in my instance of an "instance" and several other errors.

• Ratnakar Tiwari suggested a footnote explaining degenerate triangles.

• Anurag Goel suggested another solution for `is_abecedarian` and sent some additional corrections. And he knows how to spell Jane Austen.

• Kelli Kratzer spotted one of the typos.

• Mark Griffiths pointed out a confusing example in Chapter 3.

• Roydan Ongie found an error in my Newton's method.

• Patryk Wolowiec helped me with a problem in the HTML version.

• Mark Chonofsky told me about a new keyword in Python 3.0.

• Russell Coleman helped me with my geometry.

• Wei Huang spotted several typographical errors.

• Karen Barber spotted the the oldest typo in the book.

• Nam Nguyen found a typo and pointed out that I used the Decorator pattern but didn't mention it by name.

• Stephane Morin sent in several corrections and suggestions. ´

Contents

Chapter 1

The way of the program

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

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The way of the program."

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

1.1 The Python programming language

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

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

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

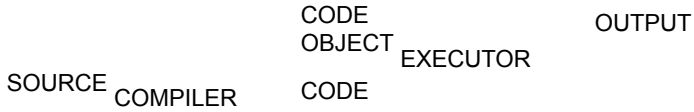
Due to these advantages, almost all programs are written in high-level languages. Low-level lan guages are used only for a few specialized applications.

Two kinds of programs process high-level languages into low-level languages: **interpreters** and

compilers. An interpreter reads a high-level program and executes it, meaning that it does what the

program says. It processes the program a little at a time, alternately reading lines and performing computations.

A compiler reads the program and translates it completely before the program starts running. In this context, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. 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 inter preter. There are two ways to use the interpreter: **interactive mode** and **script mode**. In interactive mode, you type Python programs and the interpreter prints the result:

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

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

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

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.

1.2 What is a program?

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

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

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

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

math: Perform basic mathematical operations like addition and multiplication.

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

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

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

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

1.3 What is debugging?

Programming is error-prone. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**.

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

1.3.1 Syntax errors

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

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

1.3.2 Runtime errors

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

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

1.3.3 Semantic errors

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

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

1.3.4 Experimental debugging

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

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

Debugging is also like an experimental science. Once you have an idea 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.

1.4 Formal and natural languages

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

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting rela tionships 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 cor rect 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 unam biguous, 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.

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

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

1.5 The first program

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

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

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

```
Hello, World!
```

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

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

1.6 Debugging

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

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 strug gling 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 com puter as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

[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*.

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

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

1.7 Glossary

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

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

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

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

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

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

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

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

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

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

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

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

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

program: A set of instructions that specifies a computation.

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

bug: An error in a program.

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

syntax: The structure of a program.

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

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

semantics: The meaning of a program.

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

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

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

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

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

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

1.8 Exercises

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

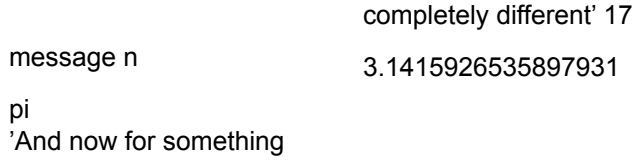
One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

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 exponen tiation, 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 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 equiv alent to `4+4+4`, we expect `'Spam'*3` to be the same as `'Spam'+'Spam'+'Spam'`, and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition has that string concatenation does not?

2.9 Comments

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

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

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

In this case, the comment appears on a line by itself. You can also put comments at the end of a line: `percentage = (minute * 100) / 60 # percentage of an hour` Everything from the `#` to the end of the line is ignored—it has no effect on the program.

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

This comment is redundant with the code and useless:

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

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

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

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

2.10 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 π/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.

2.11 Glossary

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

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

integer: A type that represents whole numbers.

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

string: A type that represents sequences of characters.

variable: A name that refers to a value.

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

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

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

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

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

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

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

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

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

value.

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

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

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.

2.12 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

Functions

3.1 Function calls

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

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

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

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

3.2 Type conversion functions

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

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

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

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

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


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

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

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

3.3 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
```

3.4 Composition

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

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

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

And even function calls:

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

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error[1].

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

3.5 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.

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, num bers 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 3.13). The body can contain any number of statements.

[1] We will see exceptions to this rule later.

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.

3.6 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 3.1 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 3.2 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?

3.7 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.

3.8 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`.

3.9 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 exam ple 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:

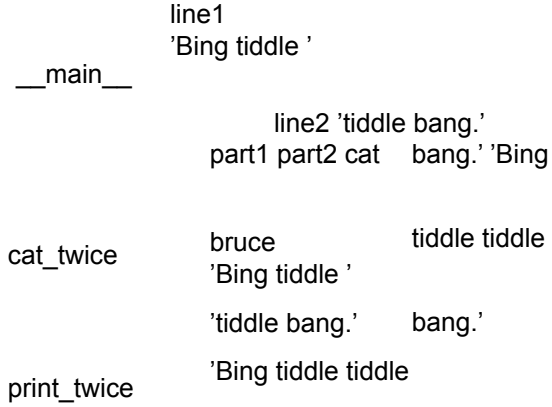
```
>>> print cat
NameError: name 'cat' is not defined
```

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

3.10 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function 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.

3.11 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'>
```

The functions we have written so far are all void. We will start writing fruitful functions in a few chapters.

3.12 Why functions?

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

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to 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.

3.13 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!

3.14 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.

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

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

fruitful function: A function that returns a value.

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

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

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

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

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

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.

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.

3.15 Exercises

Exercise 3.3 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 3.4 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 3.5 This exercise[2] 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] Based on an exercise in Oualline, *Practical C Programming, Third Edition*, O'Reilly (1997)

 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`.

Chapter 4

Case study: interface design

4.1 TurtleWorld

To accompany this book, I have written a suite of modules called Swampy. One of these modules is TurtleWorld, which provides a set of functions for drawing lines by steering turtles around the screen.

You can download Swampy from `thinkpython.com/swampy`; follow the instructions there to install Swampy on your system.

Move into the directory that contains `TurtleWorld.py`, create a file named `polygon.py` and type in the following code:

```
from TurtleWorld import *

world = TurtleWorld()
bob = Turtle()
print bob

wait_for_user()
```

The first line is a variation of the `import` statement we saw before; instead of creating a module object, it imports the functions from the module directly, so you can access them without using dot notation.

The next lines create a TurtleWorld assigned to `world` and a Turtle assigned to `bob`. Printing `bob` yields something like:

```
<TurtleWorld.Turtle instance at 0xb7bfbf4c>
```

This means that `bob` refers to an **instance** of a Turtle as defined in module `TurtleWorld`. In this context, "instance" means a member of a set; this Turtle is one of the set of possible Turtles.

`wait_for_user` tells TurtleWorld to wait for the user to do something, although in this case there's not much for the user to do except close the window.

TurtleWorld provides several turtle-steering functions: `fd` and `bk` for forward and backward, and `lt` and `rt` for left and right turns. Also, each Turtle is holding a pen, which is either down or up; if the

pen is down, the Turtle leaves a trail when it moves. The functions `pu` and `pd` stand for "pen up" and "pen down."

To draw a right angle, add these lines to the program (after creating `bob` and before calling `wait_for_user`):

```
fd(bob, 100)
rt(bob)
fd(bob, 100)
```

The first line tells `bob` to take 100 steps forward. The second line tells him to turn right.

When you run this program, you should see `bob` move east and then south, leaving two line segments behind.

Now modify the program to draw a square. Don't go on until you've got it working!

4.2 Simple repetition

Chances are you wrote something like this (leaving out the code that creates TurtleWorld and waits for the user):

```
fd(bob, 100)
lt(bob)
```

```
fd(bob, 100)
lt(bob)

fd(bob, 100)
lt(bob)

fd(bob, 100)
```

We can do the same thing more concisely with a `for` statement. Add this example to `polygon.py` and run it again:

```
for i in range(4):
    print 'Hello!'
```

You should see something like this:

```
Hello!
Hello!
Hello!
Hello!
```

This is the simplest use of the `for` statement; we will see more later. But that should be enough to let you rewrite your square-drawing program. Don't go on until you do.

Here is a `for` statement that draws a square:

```
for i in range(4):
    fd(bob, 100)
    lt(bob)
```


The syntax of a `for` statement is similar to a function definition. It has a header that ends with a colon and an indented body. The body can contain any number of statements.

A `for` statement is sometimes called a **loop** because the flow of execution runs through the body and then loops back to the top. In this case, it runs the body four times.

This version is actually a little different from the previous square-drawing code because it makes another left turn after drawing the last side of the square. The extra turn takes a little more time, but it simplifies the code if we do the same thing every time through the loop. This version also has the effect of leaving the turtle back in the starting position, facing in the starting direction.

4.3 Exercises

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

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

1. Write a function called `square` that takes a parameter named `t`, which is a turtle. It should use the turtle to draw a square.

   Write a function call that passes `bob` as an argument to `square`, and then run the program again.

2. Add another parameter, named `length`, to `square`. Modify the body so length of the sides is `length`, and then modify the function call to provide a second argument. Run the program again. Test your program with a range of values for `length`.

3. The functions `lt` and `rt` make 90-degree turns by default, but you can provide a second argument that specifies the number of degrees. For example, `lt(bob, 45)` turns `bob` 45 degrees to the left.

   Make a copy of `square` and change the name to `polygon`. Add another parameter named `n` and modify the body so it draws an n-sided regular polygon. Hint: The exterior angles of an n-sided regular polygon are 360.0/*n* degrees.

4. Write a function called `circle` that takes a turtle, `t`, and radius, `r`, as parameters and that draws an approximate circle by invoking `polygon` with an appropriate length and number of sides. Test your function with a range of values of `r`.

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

   Another hint: if `bob` is too slow for you, you can speed him up by changing `bob.delay`, which is the time between moves, in seconds. `bob.delay = 0.01` ought to get him moving.

5. Make a more general version of `circle` called `arc` that takes an additional parameter `angle`, which determines what fraction of a circle to draw. `angle` is in units of degrees, so when `angle=360`, `arc` should draw a complete circle.

4.4 Encapsulation

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

```
def square(t):
    for i in range(4):
        fd(t, 100)
        lt(t)

square(bob)
```

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

Inside the function, `t` refers to the same turtle `bob` refers to, so `lt(t)` has the same effect as `lt(bob)`. So why not call the parameter `bob`? The idea is that `t` can be any turtle, not just `bob`, so you could create a second turtle and pass it as an argument to `square`:

```
ray = Turtle()
square(ray)
```

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

4.5 Generalization

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

```
def square(t, length):
    for i in range(4):
        fd(t, length)
        lt(t)

square(bob, 100)
```

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

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

```
def polygon(t, n, length):
    angle = 360.0 / n
    for i in range(n):
        fd(t, length)
        lt(t, angle)

polygon(bob, 7, 70)
```


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

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

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

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

4.6 Interface design

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

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

The first line computes the circumference of a circle with radius `r` using the formula $2\pi r$. Since we use `math.pi`, we have to import `math`. By convention, `import` statements are usually at the beginning of the script.

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

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

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

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

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

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

```
polygon(t, n, length)
```

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

4.7 Refactoring

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

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

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

    for i in range(n):
        fd(t, step_length)
        lt(t, step_angle)
```

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

```
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

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

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

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

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

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

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

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

4.8 A development plan

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

1. Start by writing a small program with no function definitions.

2. Once you get the program working, encapsulate it in a function and give it a name.

3. Generalize the function by adding appropriate parameters.

4. Repeat steps 1–3 until you have a set of working functions. Copy and paste working code to avoid retyping (and re-debugging).

5. Look for opportunities to improve the program by refactoring. For example, if you have similar code in several places, consider factoring it into an appropriately general function.

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

4.9 docstring

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

```
def polyline(t, length, n, angle):
"""Draw n line segments with the given length and
 angle (in degrees) between them. t is a turtle.
    """
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

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

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

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

4.10 Debugging

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

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

These requirements are called **preconditions** because they are supposed to be true before the func tion starts executing. Conversely, conditions at the end of the function are **postconditions**. Postcon ditions include the intended effect of the function (like drawing line segments) and any side effects (like moving the Turtle or making other changes in the World).

Preconditions are the responsibility of the caller. If the caller violates a (properly documented!)

precondition and the function doesn't work correctly, the bug is in the caller, not the function. How ever, for purposes of debugging it is often a good idea for functions to check their preconditions rather than assume they are true. If every function checks its preconditions before starting, then if something goes wrong, you will know which function to blame.

4.11 Glossary

instance: A member of a set. The TurtleWorld in this chapter is a member of the set of Turtle Worlds.

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

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

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

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

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

development plan: A process for writing programs.

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

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

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

4.12 Exercises

Exercise 4.1 Download the code in this chapter from `thinkpython.com/code/polygon.py`.

1. Write appropriate docstrings for `polygon`, `arc` and `circle`.

2. Draw a stack diagram that shows the state of the program while executing `circle(bob, radius)`. You can do the arithmetic by hand or add `print` statements to the code.

3. The version of `arc` in Section 4.7 is not very accurate because the linear approximation of the circle is always outside the true circle. As a result, the turtle ends up a few units away from the correct destination. My solution shows a way to reduce the effect of this error. Read the code and see if it makes sense to you. If you draw a diagram, you might see how it works.

Exercise 4.2 Write an appropriately general set of functions that can draw flowers like this:

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

Exercise 4.3 Write an appropriately general set of functions that can draw shapes like this:

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

Exercise 4.4 The letters of the alphabet can be constructed from a moderate number of basic elements, like vertical and horizontal lines and a few curves. Design a font that can be drawn with a minimal number of basic elements and then write functions that draw letters of the alphabet.

You should write one function for each letter, with names draw_a, draw_b, etc., and put your functions in a file named letters.py. You can download a "turtle typewriter" from thinkpython.com/code/typewriter.py to help you test your code.

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

Chapter 5

Conditionals and recursion

5.1 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.

5.2 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 `=>`.

5.3 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).

5.4 Conditional execution

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

```
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!
```

5.5 Alternative execution

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

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

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

5.6 Chained conditionals

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

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

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

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

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.

5.7 Nested conditionals

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

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

The outer conditional contains two branches. The first branch contains a simple statement. The sec ond 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** be come 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.'
```

5.8 Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):
    if n <= 0:
        print 'Blastoff!'
    else:
        print n
        countdown(n-1)
```

If `n` is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs `n` and then calls a function named `countdown`—itself—passing `n-1` as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```


The execution of `countdown` begins with `n=3`, and since `n` is greater than 0, it outputs the value 3, and then calls itself...

> The execution of `countdown` begins with `n=2`, and since `n` is greater than 0, it outputs the value 2, and then calls itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is greater than 0, it outputs the value 1, and then calls itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is not greater than 0, it outputs the word, “Blastoff!” and then returns.
> >
> > The `countdown` that got `n=1` returns.
>
> The `countdown` that got `n=2` returns.

The `countdown` that got `n=3` returns.

And then you’re back in `__main__`. So, the total output looks like this:

```
3
2
```

```
1
Blastoff!
```

A function that calls itself is **recursive**; the process is called **recursion**.

As another example, we can write a function that prints a string `n` times.

```
def print_n(s, n):
    if n <= 0:
        return
    print s
    print_n(s, n-1)
```

If `n <= 0` the `return` statement exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

The rest of the function is similar to `countdown`: if `n` is greater than 0, it displays `s` and then calls itself to display `s` $n-1$ additional times. So the number of lines of output is `1 + (n - 1)`, which adds up to `n`.

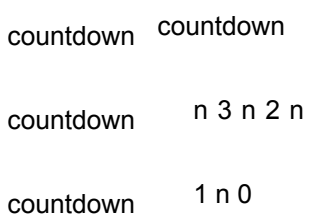
For simple examples like this, it is probably easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

5.9 Stack diagrams for recursive functions

In Section 3.10, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a new function frame, which contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

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



As usual, the top of the stack is the frame for `__main__`. It is empty because we did not create any variables in `__main__` or pass any arguments to it.

The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where `n=0`, is called the **base case**. It does not make a recursive call, so there are no more frames.

Draw a stack diagram for `print_n` called with `s = 'Hello'` and `n=2`.

Write a function called `do_n` that takes a function object and a number, `n`, as arguments, and that calls the given function `n` times.

5.10 Infinite recursion

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

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

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                .
                .
                .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1000 `recurse` frames on the stack!

5.11 Keyboard input

The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.

Python provides a built-in function called `raw_input` that gets input from the keyboard[1]. 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.

```
>>> 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`:

```
>>> 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.

5.12 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:

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

- 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 mes sage 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[2]:

```
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.

5.13 Glossary

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

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

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

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.

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

base case: A conditional branch in a recursive function that does not make a recursive call.

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

5.14 Exercises

Exercise 5.1 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 5.2 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[3]."

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.

The following exercises use TurtleWorld from Chapter 4:

Exercise 5.3 Read the following function and see if you can figure out what it does. Then run it (see the examples in Chapter 4).

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    fd(t, length*n)
    lt(t, angle)
    draw(t, length, n-1)
    rt(t, 2*angle)
    draw(t, length, n-1)
    lt(t, angle)
    bk(t, length*n)
```

Exercise 5.4 The Koch curve is a fractal that looks something like this:

To draw a Koch curve with length *x*, all you have to do is

1. Draw a Koch curve with length *x*/3.

2. Turn left 60 degrees.

3. Draw a Koch curve with length *x*/3.

4. Turn right 120 degrees.

5. Draw a Koch curve with length *x*/3.

6. Turn left 60 degrees.

7. Draw a Koch curve with length *x*/3.

The only exception is if *x* is less than 3. In that case, you can just draw a straight line with length *x*.

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

1. Write a function called `koch` that takes a turtle and a length as parameters, and that uses the turtle to draw a Koch curve with the given length.

2. Write a function called `snowflake` that draws three Koch curves to make the outline of a snowflake.

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

3. The Koch curve can be generalized in several ways. See `wikipedia.org/wiki/Koch_snowflake` for examples and implement your favorite.

Chapter 6

Fruitful functions

6.1 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 6.1 Write a `compare` function that returns `1` if `x > y`, `0` if `x == y`, and `-1` if `x < y`.

6.2 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 ex perience, 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 consoli date multiple statements into compound expressions, but only if it does not make the program difficult to read.

Exercise 6.2 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.

6.3 Composition

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

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

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. 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))
```

6.4 Boolean functions

Functions can return booleans, which is often convenient for hiding complicated tests inside func tions. 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 6.3 Write a function `is_between(x, y, z)` that returns `True` if $x \leq y \leq z$ or `False` otherwise.

6.5 More recursion

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

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first com puter scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. For a more complete (and accurate) discussion of the Turing Thesis, I recommend Michael Sipser's book *Introduction to the Theory of Computation*.

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

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

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol !, you might get something like this:

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

This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n-1$.

So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, 3! equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that `factorial` takes an integer:

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

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

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

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n-1$ and then multiply it by n:

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

```
    return result
```

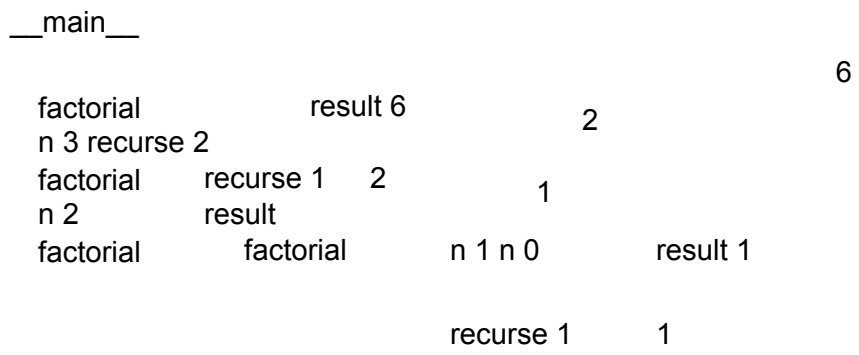
The flow of execution for this program is similar to the flow of `countdown` in Section 5.8. If we call `factorial` with the value 3:

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

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

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

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



The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of `n` and `recurse`.

In the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them does not execute.

6.6 Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become labyrinthine. An alternative is what I call the "leap of faith." When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the right result.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `math.cos` or `math.exp`, you don't examine the bodies of those functions. You just assume that they work because the people who wrote the built-in functions were good programmers.

The same is true when you call one of your own functions. For example, in Section 6.4, we wrote a function called `is_divisible` that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—by examining the code and testing—we can use the function without looking at the body again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (yields the correct result) and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

Of course, it's a bit strange to assume that the function works correctly when you haven't finished writing it, but that's why it's called a leap of faith!

6.7 One more example

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`, which has the following definition[1]:

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

fibonacci(0) = 0

fibonacci(1) = 1

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

Translated into Python, it looks like this:

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

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

6.8 Checking types

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

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

It looks like an infinite recursion. But how can that be? There is a base case—when `n == 0`. But if `n` is not an integer, we can *miss* the base case and recurse forever.

In the first recursive call, the value of `n` is 0.5. In the next, it is -0.5. From there, it gets smaller (more negative), but it will never be 0.

We have two choices. We can try to generalize the `factorial` function to work with floating-point numbers, or we can make `factorial` check the type of its argument. The first option is called the gamma function[2]and it's a little beyond the scope of this book. So we'll go for the second.

We can use the built-in function `isinstance` to verify the type of the argument. While we're at it, we can also make sure the argument is positive:

```
def factorial (n):
    if not isinstance(n, int):
        print 'Factorial is only defined for integers.'
```

```
    return None
elif n < 0:
print 'Factorial is only defined for positive integers.'
    return None
elif n == 0:
    return 1
else:
    return n * factorial(n-1)
```

[2] See wikipedia.org/wiki/Gamma_function.

The first base case handles nonintegers; the second catches negative integers. In both cases, the program prints an error message and returns `None` to indicate that something went wrong:

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is only defined for positive integers.
None
```

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

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

6.9 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 6.2).

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. For example, here is a version of `factorial` with print statements:

```
def factorial(n):
    space = ' ' * (4 * n)
    print space, 'factorial', n
    if n == 0:
        print space, 'returning 1'
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
```

```
    print space, 'returning', result
    return result
```


`space` is a string of space characters that controls the indentation of the output. Here is the result of `factorial(5)` :

```
                    factorial 5
                factorial 4
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24
                    returning 120
```

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but a little bit of scaffolding can save a lot of debugging.

6.10 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.

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 cir cumstances that might cause an error.

6.11 Exercises

Exercise 6.4 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):
```