
# core PYTHON programming

- **New to Python?** This is *the* guide to Python development!
- **Covers** core Python features *and* advanced topics
- **Learn** about regular expressions, multithreaded programming, Web development, GUI development—and more
- **Also includes** important new features found in the newest Python releases
- **CD-ROM:** Complete Python distributions (source code, documentation, and various binaries), plus all examples in the book

WESLEY J. CHUN


Core Python Programming

Wesley J. Chun
Publisher: Prentice Hall PTR
First Edition December 14, 2000
ISBN: 0-13-026036-3, 816 pages

# Review

New to Python? This is the developer's guide to Python development!

Learn the core features of Python as well as advanced topics such as regular expressions, multithreaded programming, Web/Internet and network development, GUI development with Tk(inter) and more

Also includes features found in the new Python 1.6 and 2.0 releases

CD-ROM: Complete Python distributions (source code, documentation, and various binaries) plus all example scripts in the book

Python is an Internet and systems programming language that is soaring in popularity in today's fast-paced software development environment, and no wonder: it's simple (yet robust), object-oriented (yet can be used as a procedural language), extensible, scalable and features an easy to learn syntax that is clear and concise. Python combines the power of a compiled object language like Java and C++ with the ease of use and rapid development time of a scripting language. In fact, its syntax is so easy to understand that you are more likely to pick it up faster than any of the other popular scripting languages in use today!

In *Core Python Programming*, Internet software engineer and technical trainer Wesley Chun provides intermediate and experienced developers all they need to know to learn Python-fast. Like all Core Series books, Core Python Programming delivers hundreds of industrial-strength code snippets and examples, all targeted at professional developers who want to leverage their existing skills! In particular, *Core Python Programming* presents numerous interactive examples that can be entered into the Python interpreter right in front of you! Finally, we present a chapter that shows you step-by-step how to extend Python using C or C++.

Python syntax and style

Development and Run-time Environments

Objects and Python memory management

Standard data types, methods, and operators

Loops and conditionals

Files and Input/Output

Exceptions and error handling

Functions, scope, arguments, and functional programming

Importing modules and module attributes

Object-oriented Programming with classes, methods, and instances

Callable Objects

Extending Python

Coverage of the Python standard module library and client-server application development includes comprehensive introductions to the following topics in Python programming:

Regular expressions

TCP/IP and UDP/IP Network programming using sockets

Operating system interface

GUI development with Tk using Tkinter

Multithreaded programming

Interactive Web/CGI/Internet applications

Executing code in a restricted environment

Inheritance, type emulation, operator overloading, and delegation in an OOP environment

Finally, we provide an introduction to the new features introduced in Python 1.6. These include Unicode string support, the new function invocation syntax which lets the caller provide a tuple of positional arguments and/or a dictionary of keyword arguments, and the new string methods. We also provide a glimpse into features that will only be found in the newer 2.0 release.

Every Core Series book:

DEMONSTRATES how to write commercial-quality code

FEATURES dozens of programs and examples!

FOCUSES on the features and functions most important to real developers

PROVIDES objective, unbiased coverage of cutting-edge technologies-no

hype!

*Core Python Programming* delivers:

Coverage of the core parts of the Python language

Real-world insights for developing Web/Internet, network, multithreaded and GUI applications

Tables and charts detailing Python modules, built-in functions, operators, and attributes

Code snippets to try live with Python's interactive interpreter, hammering the concepts home

Extensive code examples-including several complete sample applications

CD-ROM includes complete Python source code and documentation distributions for Unix/Linux along with binaries for Windows and Macintosh platforms plus source code for all examples in the book.

**Library of Congress Cataloging-in-Publication Date**

Chun, Wesley

Core python / Wesley. Chun.

p. cm.

Includes bibliographical references and index.

ISBN 0-13-026036-3

1. Python (Computer program language) I. Title

QA76.73.P98 C48 2000

005.13'3--dc21 00-047856

### Copyright Information



Prentice-Hall, Inc

Upper Saddle River, NJ 07458

The publisher offers discounts on this book when ordered in bulk quantities.

For more information, contact

Corporate Sales Department,

Prentice Hall PTR

One Lake Street

Upper Saddle River, NJ 07458

Phone: 800-382-3419; FAX: 201-236-7141

E-mail (Internet): corpsales@prenhall.com



10 9 8 7 6 5 4 3 2 1

Prentice-Hall International (UK) Limited, London

Prentice-Hall of Australia Pty. Limited, Sydney

Prentice-Hall Canada Inc., Toronto

Prentice-Hall Hispanoamericana, S.A., Mexico

Prentice-Hall of India Private Limited, New Delhi

Prentice-Hall of Japan, Inc., Tokyo

Pearson Education P.T.E., Ltd.

To my parents,

who taught me that everybody is different.

And to my wife,

who *lives* with someone who is different.

## Table of Contents

# Welcome to Python!

Welcome to the wonderful world of Python! As a professional or student with working knowledge of another high-level programming language, this text was made for you in your efforts to jump straight into Python with as little overhead as possible. The goal of this book is to provide text that flows in a conversational style littered with examples to highlight your path towards Python programming.

At the time of publication, Python 2.0 was just released, so you will definitely have the latest and greatest. The supplementary CD-ROM has the three most recent versions of Python: 1.5.2, 1.6, and 2.0, not to mention the most recent release of the Java version of the Python interpreter, JPython (a.k.a. Jython).

## Style:Technical, Yet Easy Reading

Rather than strictly a "beginners'" book or a pure, hard-core computer science reference book, my instructional experience indicates that an easy-to-read, yet technically-oriented book serves our purpose the best, and that is to get you up-to-speed on Python as quickly as possible, so that you can apply it to your tasks *post haste.* We will introduce concepts coupled with appropriate examples to expedite the learning process. At the end of each chapter you will find numerous exercises to reinforce some of the concepts and ideas acquired in your reading.

After the obligatory introduction to Python, but before heading to the core of the language, we take a "quick plunge" into Python with the "Getting Started" chapter. The intention of this chapter is for those who wish to temporarily dispense of formal reading and get their hands dirty with Python immediately. If you do not wish to travel this path, you may proceed as normal to the next set of chapters, an introduction to Python objects. Python's primitive data types, numbers, strings, lists, tuples, and dictionaries make up the next three chapters.

Python's error-handling capability is extremely useful to both the programmer and the user, and we address that topic in a separate chapter. Finally, the largest parts of the Python "core" we cover will be functions, modules, and classes… each in its own chapter. The final chapter of the text provides insight on how Python may be extended. The last section of the book is a mini-reference guide in the appendix. There we spill the beans on the core modules of the standard library, highlight the operators and built-in operators and functions for the Python types, provide solutions to selected exercises, and conclude with a small glossary of terms.

## Author's Experience with Python

I discovered Python several years ago at a company called Four11. At the time, the company had one major product, the Four11.com White Page directory service. Python was being used to design the Rocketmail web-based email service that would eventually one day evolve into what is Yahoo!Mail today.

In addition to the use of C++, much of the controlling software and web front-end were done completely in Python. I participated in work done on the Yahoo!Mail address book and spellchecker. Since then, Python's appearance has spread to other Yahoo! sites, including People Search, Yellow Pages, and Maps and Driving Directions, just to name a few.

Although Python was new to me at the time, it was fairly easy to pick up; much simpler than other languages that I have learned in the past. The scarcity of the number of textbooks at the time led me to primarily use the Library Reference and Quick Reference Guide as my tools in learning, and also led to the motivation of the book you are reading right now.

## Book Contents

This book is divided into two main sections. The first part, taking up about two-thirds of the text, gives you treatment of the "core" part of the language, and the second part provides a set of various advanced topics to show what you can build using Python.

Python is everywhere—sometimes it is amazing to discover who is using Python and what they are doing with it—and although we would have loved to produce additional chapters on such topics as Databases (RDBMSs, SQL, etc.), CGI Processing with HTMLgen, XML, Numerical/Scientific Processing, Visual and Graphics Image Manipulation, and Zope, there simply wasn't enough time to develop these topics into their own chapters. However, we are certainly glad that we were at least able to provide you with a good introduction to many of the key areas of Python development.

Here is a chapter-by-chapter guide:

Part I: Core Python

## Chapter 1—Welcome to Python!

We begin by introducing Python to you, its history, features, benefits, etc., as well as how to obtain and install Python on your system.

## Chapter 2—Getting Started

If you are an experienced programmer and just want to see "how it's done" in Python, this is the right place to go. We introduce the basic Python concepts and statements, and because many of these would be familiar to you, you would simply learn the proper

syntax in Python and can get started right away on your projects without sacrificing too much reading time.

## Chapter 3—Syntax and Style

This section gives you a good overview of Python's syntax as well as style hints. You will also be exposed to Python's keywords and its memory management ability. Your first Python application will be presented at the end of the chapter to give you an idea of what real Python code looks like.

## Chapter 4—Python Objects

This chapter introduces Python objects. In addition to generic object attributes, we will show you all of Python's data types and operators, as well as show you different ways to categorize the standard types. Built-in functions that apply to most Python objects will also be covered.

## Chapter 5—Numbers

Python has four numeric types: regular or "plain" integers, long integers, floating point real numbers, and complex numbers. You will learn about all four here, as well as the operators and built-in functions that apply to numbers.

## Chapter 6—Sequences: Strings, Lists, and Tuples

Your first meaty chapter will expose you to all of Python's powerful sequence types: strings, lists, and tuples. We will show you all the built-in functions, methods, and special features, which apply to each type as well as all their operators.

## Chapter 7—Dictionaries

Dictionaries are Python's mapping or hashing type. Like other data types, dictionaries also have operators and applicable built-in functions and methods.

## Chapter 8—Conditionals and Loops

Like many other high-level languages, Python supports loops such as for and while, as well as if statements (and related). Python also has a built-in function called `range()` which enables Python's for loop to behave more like a traditional counting loop rather than the foreach iterative type loop that it is.

## Chapter 9—Files and Input/Output

In addition to standard file objects and input/output, this chapter introduces you to file system access, file execution, and persistent storage.

## Chapter 10—Errors and Exceptions

One of Python's most powerful constructs is its exception handling ability. You can see a full treatment of it here, instruction on how to raise or throw exceptions, and more importantly, how to create your own exception classes.

## Chapter 11—Functions

Creating and calling functions are relatively straightforward, but Python has many other features that you will find useful, such as default arguments, named or keyword arguments, variable-length arguments, and some functional programming constructs. We also dip into variable scope and recursion briefly.

## Chapter 12—Modules

One of Python's key strengths is in its ability to be extended. This feature allows for "plug-n-play" access as well as promotes code reuse. Applications written as modules can be imported for use by other Python modules with a single line of code. Furthermore, multiple module software distribution can be simplified by using packages.

## Chapter 13—Classes and OOP

Python is a fully object-oriented programming language and was designed that way from the beginning. However, Python does not require you to program in such a manner—you may continue to develop structural/procedural code as you like, and can transition to "OO" programming anytime you are ready to take advantage of its benefits. Likewise, this chapter is here to guide you through the concepts as well as advanced topics, such as operator overloading, customization, and delegation.

## Chapter 14—Execution Environment

The term "execution" can mean many different things, from callable and executable objects to running other programs (Python or otherwise). We discuss these topics in this chapter, as well as limited restricted execution and different ways of terminating execution.

# Part II: Advanced Topics

## Chapter 15—Regular Expressions

Regular expressions are a powerful tool used for pattern matching, extracting, and search-and-replace functionality. Learn about them here.

## Chapter 16—Network Programming with Sockets

So many applications today need to be network-oriented. You have to start somewhere. In this chapter, you will learn to create clients and servers, using TCP/IP and UDP/IP.

## Chapter 17—Multithreaded Programming

Multithreaded programming is a powerful way to improve the execution performance of many types of application. This chapter ends the drought of written documentation on how to do threads in Python by explaining the concepts and showing you how to correctly build a Python multithreaded application.

## Chapter 18—GUI Programming with Tkinter

Based on the Tk graphical toolkit, Tkinter is Python's default GUI development module. We introduce Tkinter to you by showing you how to build simple sample GUI applications (say that 10 times, real fast!). One of the best ways to learn is to copy, and by building on top of some of these applications, you will be on your way in no time. We conclude the chapter by presenting a more complex example.

## Chapter 19—Web Programming

Web programming using Python takes three main forms… Web clients, Web servers, and the popular Common Gateway Interface applications which help Web servers deliver dynamically-generated Web pages. We will cover them all in this chapter: simple and advanced Web clients and CGI applications, as well as how to build your own Web server.

## Chapter 20—Extending Python

We mentioned earlier how powerful it is to have the ability to reuse code and extend the language. In pure Python, these extensions are modules, but you can also develop lower-level code in C, C++, or Java, and interface those with Python in a seamless fashion. Writing your extensions in a lower-level programming language gives you added performance and some security (because the source code does not have to be revealed). This final chapter of the book walks you step-by-step through the extension building process.

## Optional Sections

Subsections or exercises marked with an asterisk ( * ) may be skipped due to their advanced or optional nature. They are usually self-contained segments that can be addressed at another time.

Those of you with enough previous programming knowledge and who have set up their Python development environments can skip the first two chapters and go straight to Chapter 2—Getting Started—where you can absorb Python into their system and be off to the races.

## Conventions

Python interpreters are available in both C and Java. To differentiate these two interpreters, the original implementation written in C is referred to as "CPython" while the native Java implementation is called "JPython." We would also like to define "Python" as the actual language definition while CPython and JPython are two interpreters that implement the language. We will refer to "`python`" as the executable file name for CPython and "`jpython`" as the executable file name for JPython.

All program output and source code are in `Courier` font. Python keywords appear in **`Courier-Bold`** font. Lines of output with three leading greater than signs, `>>>,` represent the Python interpreter prompt.

"Core Notes" are highlighted with this logo.

"Core Style" notes are highlighted with this logo.

"Core Module" notes are highlighted with this logo.

New features to Python are highlighted with this logo. The version these features first appeared in Python is given inside the logo.

## Book Support

I welcome any and all feedback:the good, the bad, and the ugly. If you have any comments, suggestions, kudos, complaints, bugs, questions… anything at all, feel free to contact me at cyberweb_consulting@yahoo.com.

You will find errata and other information at the book's Web site located on the Python Starship:

http://starship.python.net/crew/wesc/cpp/

# Acknowledgements

The first thanks belongs to Guido van Rossum, without whom this text would not even exist. With Python, Guido has created a veritable "holy grail" of languages which is an

"oh so perfect" tool in so many fields which involve programming, not to mention being a pleasure to use.

I would also like to express hearty congratulations and a warm thanks to all technical and non-technical, official and non-official, reviewers involved in this project. Without you, this text would have never been completed. In no particular order, you are presented in the following table. In particular, I'd like to recognize Dowson Tong, Dr. Candelaria de Ram, and Jim Ahlstrom for their numerous nitpicks and helpful comments throughout the entire text (you must be tired of my writing by now!); Dr. Cay Horstmann, Java guru and editor of Prentice Hall's Core series for his up-front and targeted remarks.

Thanks goes to my students at UC Santa Cruz Extension, who had to not only endure an incomplete and buggy version of this text, but also all the homework they endured in my Python programming courses. Thanks also goes to my Program Assistant, Ezequiel Jaime, who helped coordinate all the logistics of the C and Python courses; and I can't leave out James P. Prior, who, as the BASIC, FORTRAN (punch cards!), 6502 Assembly, and Pascal instructor to many of us at Pinole Valley High School, encouraged us to pick up the art of programming as well as a wry and punishing sense of humor.

Why am I writing this book? Because my thesis advisors at UC Santa Barbara, Louise Moser and P. Michael Melliar-Smith, wanted grad students who

**Table Team of Technical Reviewers**

| *Name* | *Affiliation* | *(no-spam) E-mail Address* |
|---|---|---|
| Guido van Rossum | creator of Python, PythonLabs | guido at python.org |
| Dowson Tong | | dtstong at yahoo.com |
| James C. Ahlstrom | Vice President Interet Corp. | jim at interet.com |
| Dr. S. Candelaria de Ram | Chief of Research and Technology, Cognizor | cognite at zianet.com |
| Cay S. Horstmann | San Jose State University | cay at horstmann.com |
| Michael Santos | Green Hills Software | michael at alpha.ece.ucsb.edu |
| Greg Ward | | gward at python.net |
| Vincent C. Rubino | Technical Yahoo!, Yahoo! | vcr at yahoo.com |
| Martijn Faassen | | faassen at vet.uu.nl |
| Emile van Sebille | | emile at fenx.com |
| Raymond Tsai | U. C. San Diego | rehmatlh at yahoo.com |
| Albert L. Anders | Principal Engineer Manage.COM | aanders at pacbell.net |
| Fredrik Lundh | | effbot at telia.com |
| Cameron Laird | Vice President Phaseit, Inc. | claird at NeoSoft.com |
| Fred L. Drake, Jr. | | fdrake at acm.org |
| Jeremy Hylton | | jeremy at alum.mit.edu |
| Steve Yoshimoto | | syosh at yahoo.com |

could write, and asked to make sure before letting me in the lab. I'm indebted to you both for not only encouraging your students to work hard and write, but write well.

Thanks to Alan Parsons, Eric Woolfson, Andrew Powell, Ian Bairnson, Stuart Elliott, David Paton, and the rest of the Project for the many years (including the year it took to write this book!) of listening pleasure and producing the most intellectual, thought-provoking, and technically sound music to have ever crossed my ears. I must also thank the many Projectologist Roadkillers for their kind words of support for my own "project" here.

The entire staff of Prentice Hall PTR, most notably my Acquisitions Editor Mark Taub, Production Editor Kathleen M. Caren, Managing Editor Lisa Iarkowski, Page Formatter Eileen Clark, as well as the rest of the staff at PHPTR have been invaluable in helping me put this project together, and allowing me to join the list of all-star authors of the Core series. Tom Post is the graphic artist behind some of the cool figures you see in the book. The ugly ones are solely my responsibility.

As I am Macintosh-challenged, I would like to thank Pieter Claerhout for providing the cool MacPython screen snapshot in the introductory chapter. I would also like to thank Albert Anders, who provided the inspiration for, as well as being the co-author of, the chapter on multithreaded programming.

Thanks also goes to Aahz for his multithreaded and direct remarks on the MT chapter (I get it now!), as well as inspiration for the Crawler in the Web programming chapter, fellow Yahoo! Jeffrey E. F. Friedl, "regexer-extraordinaire," who gave me valuable feedback for the Regular Expressions chapter, and Fredrik Lundh, another regex luminary and Tk(inter) expert, for valuable comments and suggestions for those corresponding chapters. Catriona (Kate) Johnston gave me wonderful newbie feedback on the Web programming chapter. I'd also like to thank David Ascher (Python expert), Reg Charney (fearless leader of the Silicon Valley chapter of the Association of C/C++ Users), Chris Tismer (Python tinkerer), and Jason Stillwell for their helpful comments.

I would also like to thank my family, friends and the Lord above, who have kept me safe and sane during this crazy period of late nights and abandonment. And finally, I would like give a big thanks to all those who believed in me (you know who you are!)—I couldn't have done it without you. Those who didn't… well, you know what you can do! :-)

W. J. Chun

Silicon Valley, CA

(it's not as much a place as it is a state of sanity)

November 2000

# Part I: CORE PYTHON

# Chapter 1. Welcome to Python!

**Chapter Topics**

What is Python, Its History and Features

Where to Obtain Python

How to Install and Run Python

Python Documentation

Comparing Python

Our introductory chapter provides some background on what Python is, where it came from, and what some of its "bullet points" are. Once we have stimulated your interest and enthusiasm, we describe how you can obtain Python and get it up and running on your system. Finally, the exercises at the end of the chapter will make you comfortable with using Python, both in the interactive interpreter and also in creating scripts and executing them.

## What Is Python?

Python is an uncomplicated and robust programming language that delivers both the power and complexity of traditional compiled languages along with the ease-of-use (and then some) of simpler scripting and interpreted languages. You'll be amazed at how quickly you'll pick up the language as well as what kind of things you can do with Python, not to mention the things that have *already* been done. Your imagination will be the only limit.

## History of Python

Work on Python began in late 1989 by Guido van Rossum, then at CWI in the Netherlands, and eventually released for public distribution in early 1991. How did it all begin? Innovative languages are usually born from one of two motivations: a large well-funded research project or general frustration due to the lack of tools that were needed at the time to accomplish mundane and/or time-consuming tasks, many of which could be automated.

At the time, van Rossum was a researcher with considerable language design experience with the interpreted language ABC, also developed at CWI, but he was unsatisfied with its ability to be developed into something more. Some of the tools he envisioned were for performing general system administration tasks, so he also wanted access to the power of system calls that were available through the Amoeba distributed operating system.

Although an Amoeba-specific language was given some thought, a generalized language made more sense, and late in 1989, the seeds of Python were sown.

## Features of Python

Although practically a decade in age, Python is still somewhat relatively new to the general software development industry. We should, however, use caution with our use of the word "relatively," as a few years seem like decades when developing on "Internet time."

When people ask, "What is Python?" it is difficult to say any one thing. The tendency is to want to blurt out all the things that you feel Python is in one breath. Python is (fill-in-the-blanks here). Just what are some of those blanks? For your sanity, we will elucidate on each here… one at a time.

### High-level

It seems that with every generation of languages, we move to a higher level. Assembly was a godsend for those who struggled with machine code, then came FORTRAN, C, and Pascal, all of which took computing to another plane and created the software development industry. These languages then evolved into the current compiled systems languages C++ and Java. And further still we climb, with powerful, system-accessible, interpreted scripting languages like Tcl, Perl, and Python. Each of these languages has higher-level data structures that reduce the "framework" development time which was once required. Useful types like Python's lists (resizeable arrays) and dictionaries (hash tables) are built into the language. Providing these crucial building blocks encourages their use and minimizes development time as well as code size, resulting in more readable code. Implementing them in C is complicated and often frustrating due to the necessities of using structures and pointers, not to mention repetitious if some forms of the same data structures require implementation for every large project. This initial setup is mitigated somewhat with C++ and its use of templates, but still involves work that may not be directly related to the application that needs to be developed.

### Object-oriented

Object-oriented programming (OOP) adds another dimension to structured and procedural languages where data and logic are discrete elements of programming. OOP allows for associating specific behaviors, characteristics, and/or capabilities with the data that they execute on or are representative of. The object-oriented nature of Python was part of its design from the very beginning. Other OO scripting languages include SmallTalk, the original Xerox PARC language that started it all, and Netscape's JavaScript.

### Scalable

Python is often compared to batch or Unix shell scripting languages. Simple shell scripts handle simple tasks. They grow (indefinitely) in length, but not truly in depth. There is little code-reusability and you are confined to small projects with shell scripts. In fact, even small projects may lead to large and unwieldy scripts. Not so with Python, where you can grow your code from project to project, add other new or existing Python elements, and reuse code at your whim. Python encourages clean code design, high-level structure, and "packaging" of multiple components, all of which deliver the flexibility, consistency, and faster development time required as projects expand in breadth and scope.

The term "scalable" is most often applied to measuring hardware throughput and usually refers to additional performance when new hardware is added to a system. We would like to differentiate this comparison with ours here, which tries to inflect the notion that Python provides basic building blocks on which you can build an application, and as those needs expand and grow, Python's pluggable and modular architecture allows your project to flourish as well as maintain manageability.

## Extensible

As the amount of Python code increases in your project, you may still be able to organize it logically due to its dual structured and object-oriented programming environments. Or, better yet, you can separate your code into multiple files, or "modules" and be able to access one module's code and attributes from another. And what is even better is that Python's syntax for accessing modules is the same for all modules, whether you access one from the Python standard library or one you created just a minute ago. Using this feature, you feel like you have just "extended" the language for your own needs, and you actually *have.*

The most critical portions of code, perhaps those hotspots that always show up in profile analysis or areas where performance is absolutely required, are candidates for extensions as well. By "wrapping" lower-level code with Python interfaces, you can create a "compiled" module. But again, the interface is exactly the same as for pure Python modules. Access to code and objects occurs in exactly the same way without any code modification whatsoever. The only thing different about the code now is that you should notice an improvement in performance. Naturally, it all depends on your application and how resource-intensive it is. There are times where it is absolutely advantageous to convert application bottlenecks to compiled code because it will decidedly improve overall performance.

This type of extensibility in a language provides engineers with the flexibility to add-on or customize their tools to be more productive, and to develop in a shorter period of time. Although this feature is self-evident in mainstream third-generation languages (3GLs) such as C, C++, and even Java, it is rare among scripting languages. Other than Python, true extensibility in a current scripting language is readily available only in the Tool Command Language (TCL). Python extensions can be written in C and C++ for CPython and in Java for JPython.

## Portable

Python is available on a wide variety of platforms (see Section 1.4), which contributes to its surprisingly rapid growth in today's computing domain. Because Python is written in C, and because of C's portability, Python is available on practically every type of system with a C compiler and general operating system interfaces.

Although there are some platform-specific modules, any general Python application written on one system will run with little or no modification on another. Portability applies across multiple architectures as well as operating systems.

## Easy-to-learn

Python has relatively few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language in a relatively short period of time. There is no extra effort wasted in learning completely foreign concepts or unfamiliar keywords and syntax. What may perhaps be new to beginners is the object-oriented nature of Python. Those who are not fully-versed in the ways of object-oriented programming (OOP) may be apprehensive about jumping straight into Python, but OOP is neither necessary nor mandatory. Getting started is easy, and you can pick up OOP and use when you are ready to.

## Easy-to-read

Conspicuously absent from the Python syntax are the usual symbols found in other languages for accessing variables, code block definition, and pattern-matching. These include: dollar signs ( $ ), semicolons ( ; ), tildes ( ~ ), etc. Without all these distractions, Python code is much more clearly defined and visible to the eyes. In addition, much to many programmers' dismay (and relief), Python does not give as much flexibility to write obfuscated code as compared to other languages, making it easier for others to understand your code faster and vice versa. Being easy-to-read usually leads to a language's being easy-to-learn, as we described above. We would even venture to claim that Python code is fairly understandable, even to a reader who has never seen a single line of Python before. Take a look at the examples in the next chapter, Getting Started, and let us know how well you fare.

## Easy-to-maintain

Maintaining source code is part of the software development lifecycle. Your software is permanent until it is replaced or obsoleted, and in the meantime, it is more likely that your code will outlive you in your current position. Much of Python's success is that source code is fairly easy-to-maintain, dependent, of course, on size and complexity. However, this conclusion is not difficult to draw given that Python is easy-to-learn and easy-to-read. Another motivating advantage of Python is that upon reviewing a script you wrote six months ago, you are less likely to get lost or require pulling out a reference book to get reacquainted with your software.

## Robust

Nothing is more powerful than allowing a programmer to recognize error conditions and provide a software handler when such errors occur. Python provides "safe and sane" exits on errors, allowing the programmer to be in the driver's seat. When Python exits due to fatal errors, a complete stack trace is available, providing an indication of where and how the error occurred. Python errors generate "exceptions," and the stack trace will indicate the name and type of exception that took place. Python also provides the programmer with the ability to recognize exceptions and take appropriate action, if necessary. These "exception handlers" can be written to take specific courses of action when exceptions arise, either defusing the problem, redirecting program flow, or taking clean-up or other maintenance measures before shutting down the application gracefully. In either case, the debugging part of the development cycle is reduced considerably due to Python's ability to help pinpoint the problem faster rather than just being on the hunt alone. Python's robustness is beneficial for both the software designer as well as for the user. There is also some accountability when certain errors occur which are not handled properly. The stack trace which is generated as a result of an error reveals not only the type and location of the error, but also in which module the erroneous code resides.

## Effective as a Rapid Prototyping Tool

We've mentioned before how Python is easy-to-learn and easy-to-read. But, you say, so is a language like BASIC. What more can Python do? Unlike self-contained and less flexible languages, Python has so many different interfaces to other systems that it is powerful enough in features and robust enough that entire systems can be prototyped completely in Python. Obviously, the same systems can be completed in traditional compiled languages, but Python's simplicity of engineering allows us to do the same thing and still be home in time for supper. Also, numerous external libraries have already been developed for Python, so whatever your application is, someone may have traveled down that road before. All you need to do is plug-'n'-play (some assembly required, as usual). Some of these libraries include: networking, Internet/Web/CGI, graphics and graphical user interface (GUI) development (Tkinter), imaging (PIL), numerical computation and analysis (NumPy), database access, hypertext (HTML, XML, SGML, etc.), operating system extensions, audio/visual, programming tools, and many others.

## A Memory Manager

The biggest pitfall with programming in C or C++ is that the responsibility of memory management is in the hands of the developer. Even if the application has very little to do with memory access, memory modification, and memory management, the programmer must still perform those duties, in addition to the original task at hand. This places an unnecessary burden and responsibility upon the developer and often provides an extended distraction.

Because memory management is performed by the Python interpreter, the application developer is able to steer clear of memory issues and focus on the immediate goal of just

creating the application that was planned in the first place. This lead to fewer bugs, a more robust application, and shorter overall development time.

### Interpreted and (Byte-) Compiled

Python is classified as an interpreted language, meaning that compile-time is no longer a factor during development. Traditionally purely interpreted languages are almost always slower than compiled languages because execution does not take place in a system's native binary language. However, like Java, Python is actually byte-compiled, resulting in an intermediate form closer to machine language. This improves Python's performance, yet allows it to retain all the advantages of interpreted languages.

**NOTE**

Python source files typically end with the `.py` extension. The source is byte-compiled upon being loaded by the interpreter or by being byte-compiled explicitly. Depending on how you invoke the interpreter, it may leave behind byte-compiled files with a `.pyc` or `.pyo` extension. You can find out more about file extensions in *Chapter 12*, Modules.

## Obtaining Python

As we alluded to earlier in Section 1.3.5, Python is available on a wide variety of platforms:

Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, et al.)

Win 9x/NT/2000 (Windows 32-bit systems)

Macintosh (PPC, 68K)

OS/2

DOS (multiple versions)

Windows 3.x

PalmOS

Windows CE

Acorn/RISC OS

BeOS

Amiga

VMS/OpenVMS

QNX

VxWorks

Psion

There are currently three contemporary versions of Python today. 1.5.2 is the most stable version, having been released over a year and a half ago. Python 1.6, recently made available to the public in early September 2000 introduces several major new features and improvements over the 1.5 series. However, 1.6 is seen as more of a transition to the new Python 2.0, which was released in mid-October 2000. Which version should you use? The answer is based on your needs and expectations.

If you don't need all the fancy new features, but do desire rock solid stability, code which is backwards-compatible with the older releases (and cohabitating with existing Python installations), and is available on the greatest number of platforms, 1.5.2 is the obvious choice.

For all new projects, those without backwards dependence on older versions or Python, and those either wanting or needing to take advantage of the most crucial new features such as Unicode support, not to mention wanting to have access to the latest and greatest, cutting edge Python technology, you should start with 2.0.

1.6 is an alternative for those migrating from 1.5.2 to 2.0 who need a migration path, but is otherwise not recommended since it was only the most current version of Python by slightly over a month's time.

## Obtaining Python

For the most up-to-date and current source code, binaries, documentation, news, etc., check either the main Python language site or the PythonLabs Web site:

```
                                        http://www.python.org
(community home page)
http://www.pythonlabs.com     (commercial home page)
```

If you do not have access to the Internet readily available, all three versions (source code and binaries) are available on the CD-ROM in the back of the book. The CD-ROM also features the complete online documentation sets viewable via offline browsing or as archive files which can be installed on hard disk. All of the code samples in the book are there as well as the Online Resources appendix section (featured as the Python "hotlist").

## Installing Python

Platforms with ready-to-install binaries require only the file download and initiation of the installation application. If a binary distribution is not available for your platform, you

need to obtain and compile the source code manually. This is not as bad an option as it may seem at first. Manually building your own binaries offers the most flexibility.

You can choose what features to put into your interpreter and which to leave out. The smaller your executable, the faster it will load and run. For example, on Unix systems, you may wish to install the GNU `readline` module. This allows you to scroll back through Python commands and use Emacs- or `vi`-like key bindings to scroll through, access, and perhaps edit previous commands. Other popular options include incorporating Tkinter so that you can build GUI applications or the threading library to create multi-threaded applications. All of the options we described can be added by editing the `Modules/Setup` file found in your source distribution.

In general, these are the steps when building your own Python interpreter:

download and extract files, customizing build files (if applicable)

run `./configure script`

```
make
```

```
make install
```

Python is usually installed in a standard location so that you can find it rather easily. On Unix machines, the executable is usually installed in `/usr/local/bin` while the libraries are in `/usr/local/lib/python1.x` where the `1.x` is the version of Python you are using.

On DOS and Windows, you will usually find Python installed in `C:\Python` or `C:\Program Files\Python.` Since DOS does not support long names like "Program Files," it is usually aliased as "`Progra~1,`" so if you are in a DOS window in a Windows system, you will have to use the short name to get to Python. The standard library files are typically installed in `C:\Program Files\Python\Lib.`

## Running Python

There are three different ways to start Python. The simplest way is by starting the interpreter interactively, entering one line of Python at a time for execution. Another way to start Python is by running a script written in Python. This is accomplished by invoking the interpreter on your script application. Finally, you can run from a graphical user interface (GUI) from within an integrated development environment (IDE). IDEs typically feature additional tools such as debuggers and text editors.

### Interactive Interpreter from the Command-line

You can enter Python and start coding right away in the interactive interpreter by starting it from the command line. You can do this from Unix, DOS, or any other system which provides you a command-line interpreter or shell window. One of the best ways to start

learning Python is to run the interpreter interactively. Interactive mode is also very useful later on when you want to experiment with specific features of Python.

### Unix

To access Python, you will need to type in the full pathname to its location unless you have added the directory where Python resides to your search path. Common places where Python is installed include `/usr/bin` and `/usr/local/bin`.

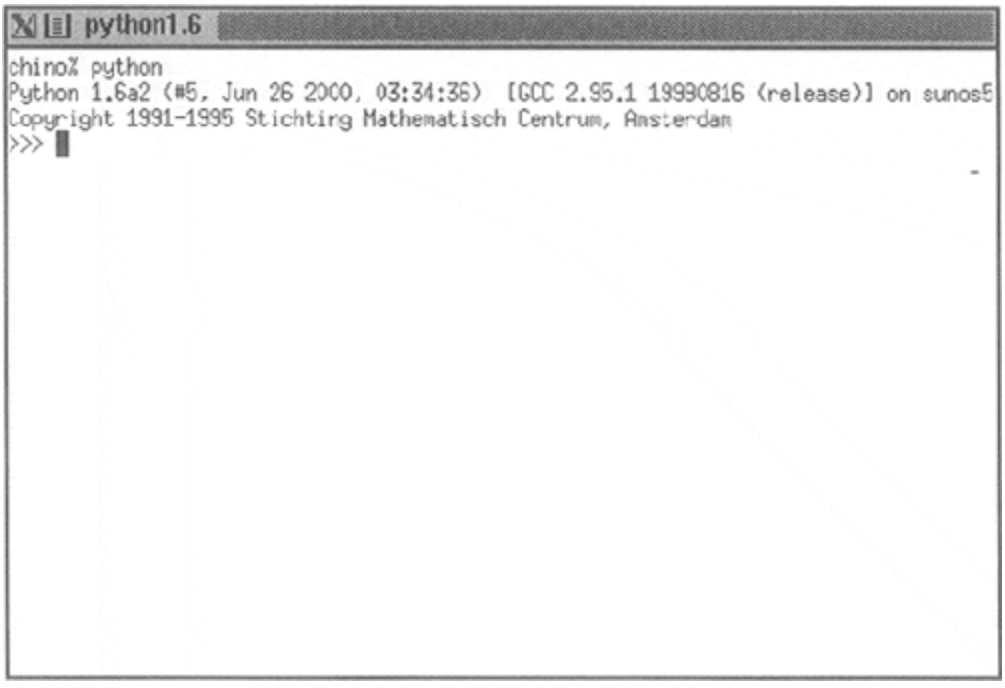
We recommend that you add Python (i.e., the executable file `python`, or `jpython` if you wish to use the Java version of the interpreter) to your search path because you do not want to have to type in the full pathname every time you wish to run interactively. Once this is accomplished, you can start the interpreter with just its name.

To add Python to your search path, simply check your login start-up scripts and look for a set of directories given to the `set path` or `PATH=` directive. Adding the full path to where your Python interpreter is located is all you have to do, followed by refreshing your shell's path variable. Now at the Unix prompt (% or $, depending on your shell), you can start the interpreter just by invoking the name `python` (or `jpython`), as in the following:

***% python***

Once Python has started, you'll see the interpreter startup message indicating version and platform and be given the interpreter prompt "`>>>`" to enter Python commands. Figure1-1 is a screen shot of what Python looks like when you start it in a Unix environment:

**Figure 1-1. Starting Python in a Unix (Solaris) Window**

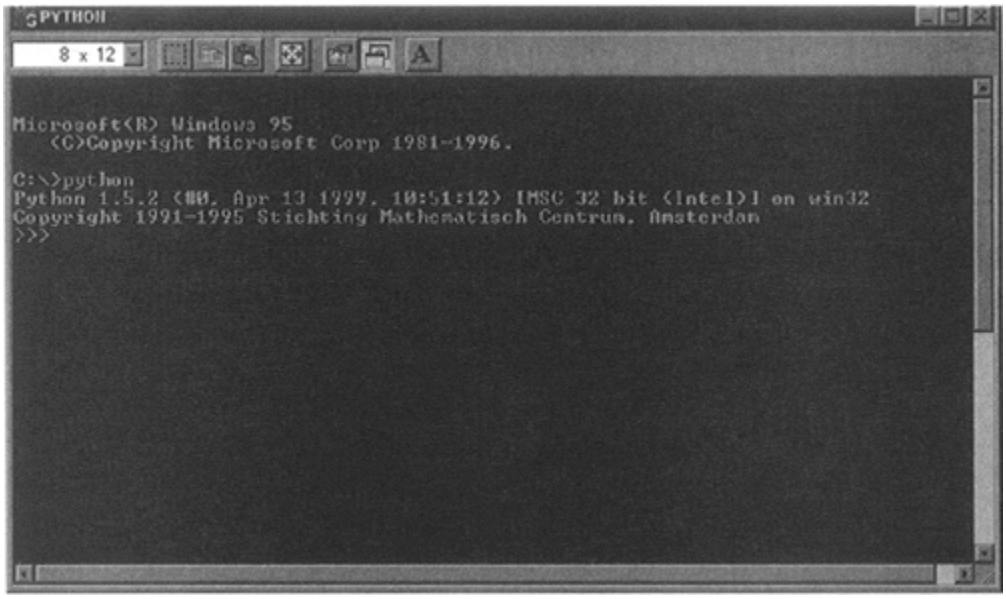


### DOS

To add Python to your search path, you need to edit the `C:\autoexec.bat` file and add the full path to where your interpreter is installed. It is usually either `C:\Python` or `C:\Program Files \Python` (or its short DOS name equivalent `C:\Progra~1\Python`). From a DOS window (either really running in DOS or started from Windows), the command to start Python is the same as Unix, python. The only difference is the prompt, which is `C:\>`.

C:> python

**Figure 1-2. Starting Python in a DOS Window**




### Command-line Options

When starting Python from the command-line, additional options may be provided to the interpreter. Here are some of the options to choose from:

| | |
|---|---|
| `-d` | provide debug output |
| `-O` | generate optimized bytecode (resulting in `.pyo` files) |
| `-S` | do not run `import site` to look for Python paths on startup |
| `-v` | verbose output (detailed trace on **`import`** statements) |
| `-X` | disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6 |
| `-c cmd` | run Python script sent in as cmd string |
| `file` | run Python script from given file (see below) |

## As a Script from the Command-line

From Unix, DOS, or any other version with a command-line interface, a Python script can be executed by invoking the interpreter on your application, as in the following:

```
C:\>python script.py
unix%python script.py
```

Most Python scripts end with a file extension of `.py`, as indicated above.

It is also possible in Unix to automatically launch the Python interpreter without explicitly invoking it from the command-line. If you are using any Unix-flavored system, you can use the shell-launching ("sh-bang") first line of your program:

```
#!/usr/local/bin/python
```

The "file path," i.e., the part that follows the "#!," is the full path location of the Python interpreter. As we mentioned before, it is usually installed in `/usr/local/bin` or `/usr/bin`. If not, be sure to get the exact pathname correct so that you can run your Python scripts. Pathnames that are not correct will result in the familiar "`Command not found`" error message.

As a preferred alternative, many Unix systems have a command named env, either installed in `/bin` or `/usr/bin`, that will look for the Python interpreter in your path. If you have env, your startup line can be changed to something like this:

```
#!/usr/bin/env python
```

`env` is useful when you either do not know exactly where the Python executable is located, or if it changes location often, yet still remains available via your directory path. Once you add the proper startup directive to the beginning of your script, it becomes directly executable, and when invoked, loads the Python interpreter first, then runs your script. As we mentioned before, Python no longer has to be invoked explicitly from the command. You only need the script name:

```
unix%
script.py
```

Be sure the file permission mode allows execution first. There should be an `'rwx'` flag for the user in the long listing of your file. Check with your system administrator if you require help in finding where Python is installed or if you need help with file permissions or the `chmod` (CHange MODe) command.

DOS does not support the auto-launching mechanism; however, Windows does provide a "file type" interface. This interface allows Windows to recognize file types based on extension names and to invoke a program to "handle" files of predetermined types. For example, if you install Python with PythonWin (see below), double-clicking on a Python script with the `.py` extension will invoke Python or PythonWin IDE (if you have it installed) to run your script.

## In an Integrated Development Environment

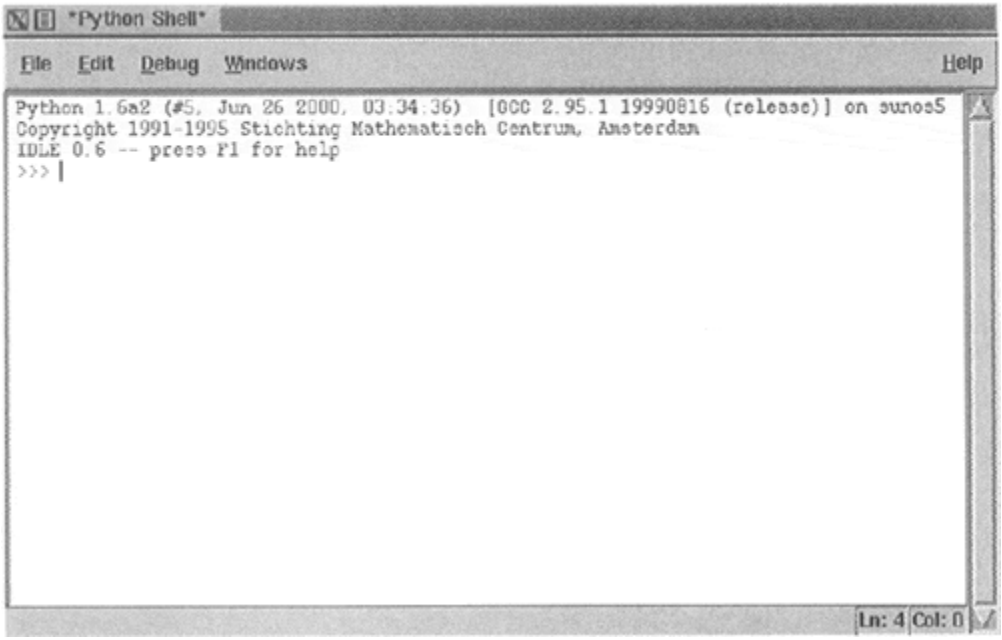
You can run Python from a graphical user interface (GUI) environment as well. All you need is a GUI application on your system that supports Python. If you have found one, chances are that it is also an IDE (integrated development environment). IDEs are more than just graphical interfaces. They typically have source code editors and trace and debugging facilities.

### Unix

IDLE is the very first Unix IDE for Python. It was also developed by Guido and made its debut in Python 1.5.2. IDLE either stands for IDE with a raised "L," as in Integrated DeveLopment Environment. Suspiciously, IDLE also happens to be the name of a Monty Python troupe member. Hmmm…. IDLE is Tkinter-based, thus requiring you to have Tcl/Tk installed on your system. Current versions of Python include a distributed minimal subset of the Tcl/Tk library so that a full install is no longer required.

You will find the `idle` executable in the `Tools` subdirectory with the source distribution. The Tk toolkit also exists on Windows, so IDLE is also available on that platform and on the Macintosh as well. A screen shot of IDLE in Unix appears in Figure1-3.

**Figure 1-3. Starting IDLE in Unix**



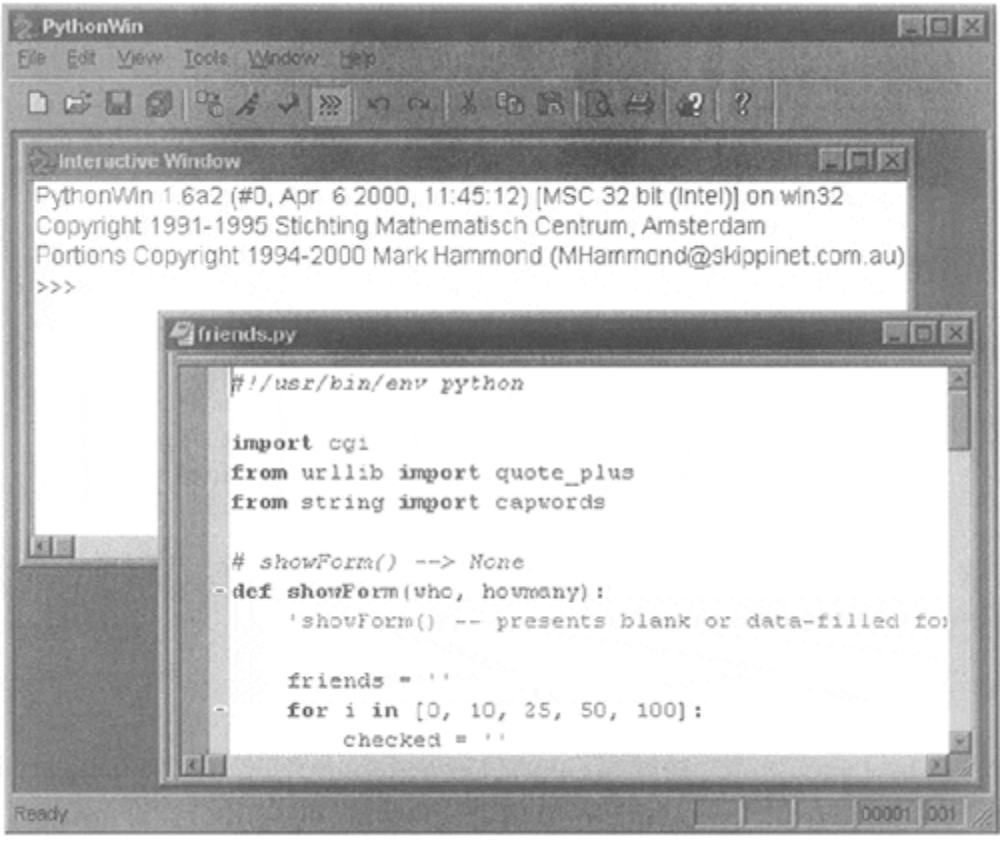

### Windows

PythonWin is the first Windows interface for Python and is an IDE with a GUI. Included with the PythonWin distribution are a Windows API, COM (Component Object Model, a.k.a. OLE [Object Linking and Embedding] and ActiveX) extensions. PythonWin itself was written to the MFC

(Microsoft Foundation Class) libraries, and it can be used as a development environment to create your own Windows applications.

PythonWin is usually installed in the same directory as Python, in its own subdirectory, `C:\Program Files\Python\Pythonwin` as the executable `pythonwin.exe`. PythonWin features a color editor, a new and improved debugger, interactive shell window, COM extensions, and more. A screen snapshot of the PythonWin IDE running on a Windows machine appears in Figure1-4.

**Figure 1-4. PythonWin Environment in Windows**

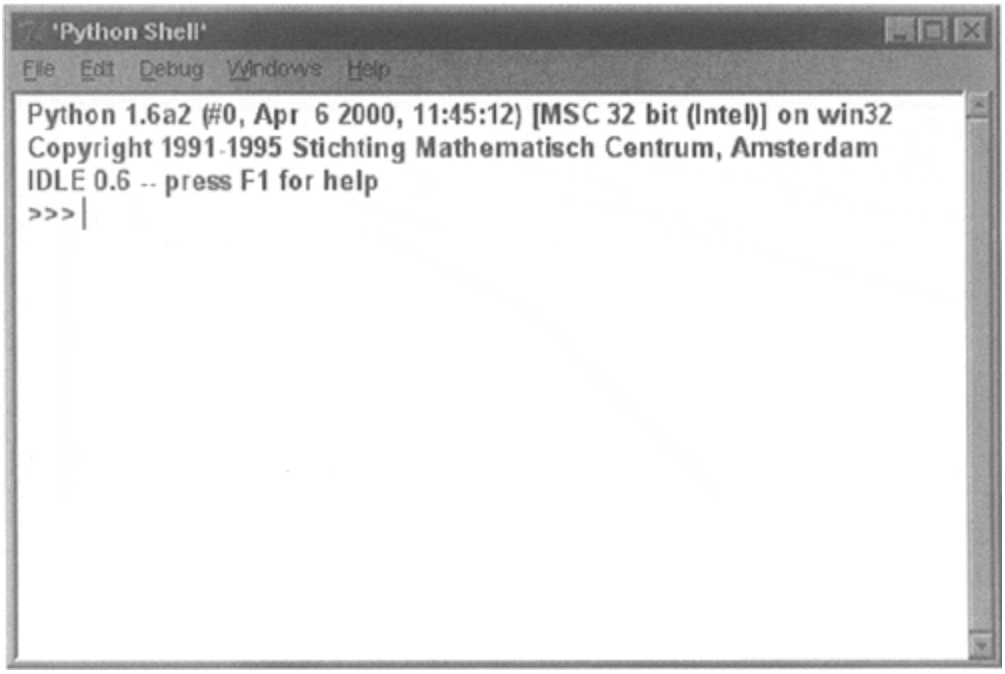



More documentation from the installed software can be found by firing up your web browser and pointing it to the following location (or wherever your PythonWin is installed):

```
file://C:/Program Files/Python/Pythonwin/readme.html
```

As we mentioned before, IDLE is also available on the Windows platform, due to the portability of Tcl/Tk and Python/Tkinter. It looks similar to its Unix counterpart (Figure1-5).

**Figure 1-5. Starting IDLE in Windows**

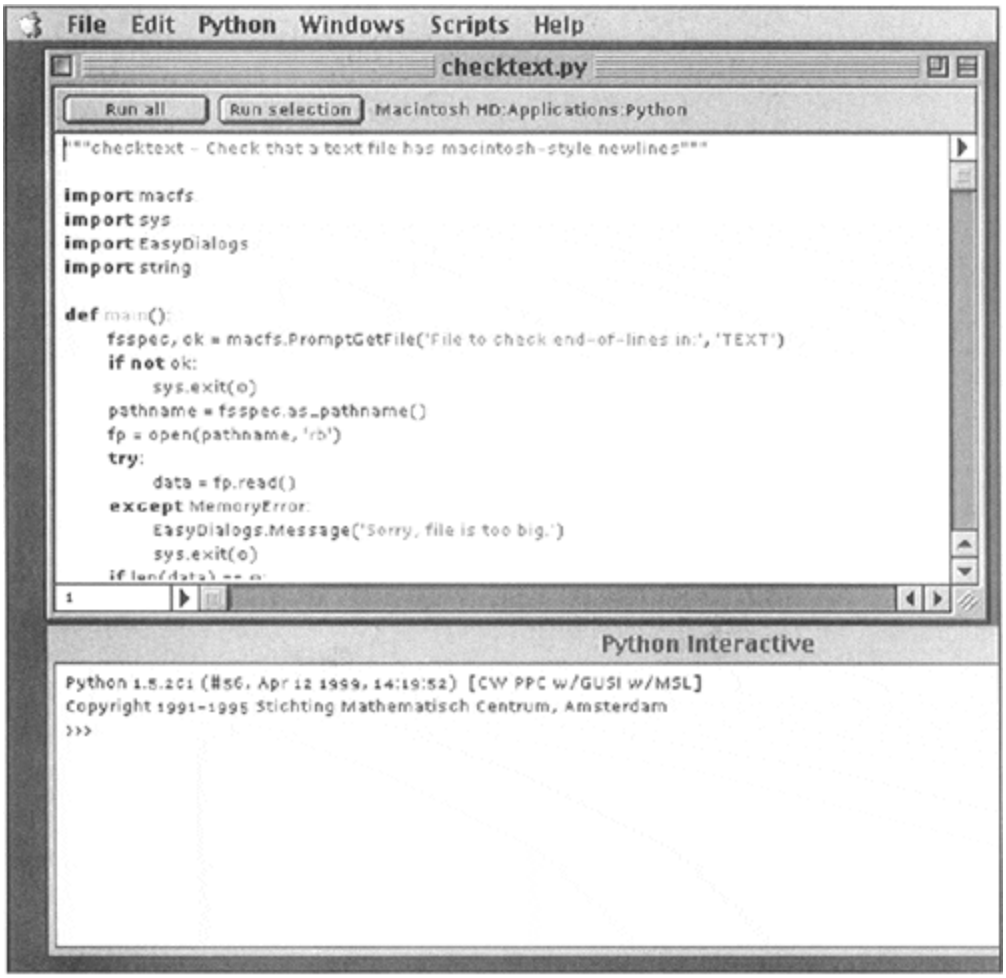



From Windows, IDLE can be found in the `Tools\idle` subdirectory of where your Python interpreter is found, usually `C:\Program Files \Python\Tools\idle.` To start IDLE from a DOS window, invoke `idle.py.` You can also invoke `idle.py` from a Windows environment, but that starts an unnecessary DOS window. Instead, double-click on `idle.pyw.`

### Macintosh

The Macintosh effort of Python is called MacPython and also available from the main website, downloadable as either MacBinary or BinHex'd files. Python source code is available as a Stuff-It archive. This distribution contains all the software you need to run Python on either the PowerPC or Motorola 68K architectures. MacPython includes an IDE, the numerical Python (NumPy) module, and various graphics modules, and the Tk windowing toolkit comes with the package, so IDLE will work on the Mac as well. Figure1-6 shows what the MacPython environment looks like. Presented in the figure below are a text window open to edit a Python script as well as a Python "shell" running the interpreter:

**Figure 1-6. Running the IDE in MacPython**




## Python Documentation

Most of the documentation that you need with Python can be found on the CD-ROM or the main website. Documentation is available for download in printable format or as hypertext HTML files for online (or offline) viewing.

If you download the Windows version of Python, the HTML documentation comes with the distribution as an install option. Be sure to leave the "Help Files" box checked if you would like to install the HTML files in your Python directory. Once the installation is complete, you may then access the Python documentation through your web browser by pointing to the link below or wherever your interpreter is installed:

```
file://C:/Program Files/Python/Doc/index.html
```

Also see the Appendix for an exhaustive list of both printed and online documentation for Python.

## Comparing Python

Python has been compared with many languages. One reason is that it provides many features found in other languages. Another reason is that Python itself is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages, to name a few. Python is a virtual "greatest hits" where van Rossum combined the features he admired most in the other languages he had studied and brought them together for our programming sanity.

However, more often than not, since Python is an interpreted language, you will find that most of the comparisons are with Perl, Java, Tcl, and JavaScript. Perl is another scripting language which goes well beyond the realm of the standard shell scripts. Like Python, Perl gives you the power of a full programming language as well as system call access.

Perl's greatest strength is in its string pattern matching ability, providing an extremely powerful regular expression matching engine. This has pushed Perl to become the de facto language for string text stream filtering, recognition, and extraction, and it is still the most popular language for developing Internet applications through web servers' Common Gateway Interface (CGI). However, Perl's obscure and overly-symbolic syntax is much more difficult to decipher, resulting in a steep learning curve that inhibits the beginner, frustrating those for whom grasping concepts is impeded by semantics. This, coupled with Perl's "feature" of providing many ways of accomplishing the same task, introduces inconsistency and factionization of developers. Finally, all too often the reference book is required reading to decipher a Perl script which was written just a mere quarter ago.

Python is often compared to Java because of their similar object-oriented nature and syntax. Java's syntax, although much simpler than C++'s, can still be fairly cumbersome, especially if you want to perform just a small task. Python's simplicity offers a much more rapid development environment that using just pure Java. One major evolution in Python's relationship with Java is the development of JPython, a Python interpreter written completely in Java. It is now possible to run Python programs with only the presence of a Java VM (virtual machine). We will mention more of JPython's advantages briefly in the following section, but for now we can tell you that in the JPython scripting environment, you can manipulate Java objects, Java can interact with Python objects, and you have access to your normal Java class libraries as if Java has always been part of the Python environment.

Tcl is another scripting language that bears some similarities to Python. Tcl is one of the first truly easy-to-use scripting languages providing the programmer extensibility as well as system call access. Tcl is still popular today and perhaps somewhat more restrictive (due to its limited types) than Python, but it shares Python's ability to extend past its original design. More importantly, Tcl is often used with its graphical toolkit partner, Tk, in developing graphical user interface (GUI) applications. Due to its popularity, Tk has been ported to Perl (Perl/Tk) and Python (Tkinter).

Python has some light functional programming (FP) constructs which likens it to languages such as Lisp or Scheme. However, it should be noted that Python is not considered an FP language; therefore, it does provide much more than what you see.

Of all the languages most often compared to Python, JavaScript bears the most resemblance. It is the most similar syntax-wise as well as also being object-oriented. Any proficient JavaScript programmer will find that picking up Python requires little or no effort. Python provides execution outside the web browser environment as well as the ability to interact with system calls and perform general system tasks commonly handled by shell scripts.

You can access a number of comparisons between Python and other languages at:

## JPython and Some Nomenclature

As we mentioned in the previous section, a Python interpreter completely (re)written in Java called JPython is currently available. Although there are still minor differences between both interpreters, they are very similar and provide a comparable startup environment.

What are the advantages of JPython? JPython…

Can run (almost) anywhere a Java Virtual Machine (JVM) can be found

Provides access to Java packages and class libraries

Furnishes a scripting environment for Java development

Enables ease-of-testing for Java class libraries

Matches object-oriented programming environments

Delivers JavaBeans property and introspection ability

Encourages Python-to-Java development (and vice versa)

Gives GUI developers access to Java AWT/Swing libraries

Utilizes Java's native garbage collector (so CPython's was not implemented)

A full treatment of JPython is beyond the scope of this text, but there is a good amount of information online. JPython is still an ongoing development project, so keep an eye out for new features.

## Exercises

1-1. *Installing Python.* Download the Python software or load it from the CD-ROM, and install it on your system.

1-2. *Executing Python.* How many different ways are there to run Python?

1-3. *Python Standard Library.*

(a) Find where the Python executables and standard library modules are installed on your system.

(b) Take a look at some of the standard library files, for example, `string.py`. It will help you get acclimated to looking at Python scripts.

> 1-4. *Interactive Execution.* Start the Python interactive interpreter. You can invoke it by typing in its full pathname or just its name (`python` or `python.exe`) if you haveinstalled its location in your search path. (You can alsouse the Python interpreter compiled in Java [`jpython or jpython.exe`] if you wish.) The startup screen should look like the ones depicted in this chapter. When you see the "`>>>`," that means the interpreter is ready to accept your Python commands.

Try entering the command for the famous Hello World! program by typing **print** "Hello World!," then exit the interpreter. On Unix systems, Ctrl-D will send the EOF signal to terminate the Python interpreter, and on DOS systems, the keypress is Ctrl-Z. Exiting from windows in graphical user environments like the Macintosh, PythonWin or IDLE on Windows, or IDLE on Unix can be accomplished by simply closing their respective windows.

1-5. *Scripting.* As a follow-up to Exercise 1–4, create "Hello World!" as a Python script that does exactly the same thing as the interactive exercise above. If you are using the Unix system, try setting up the automatic startup line so that you can run the program without invoking the Python interpreter.

1-6. *Scripting.* Create a script that displays your name, age, favorite color, and a bit about you (background, interests, hobbies, etc.) to the screen using the **`print`** statement.

# Chapter 2. Getting Started

This "quick start" section is intended to "flash" Python to you so that any constructs recognized from previous programming experience can be used for your immediate needs. The details will be spelled out in succeeding chapters, but a high-level tour is one fast and easy way to get you into Python and show you what it has to offer. The best way to follow along is to bring up the Python interpreter in front of you and try some of these examples, and at the same time you can experiment on your own.

We introduced how to start up the Python interpreter in Chapter 1 as well as in the exercises (problem 1-4). In all interactive examples, you will see the Python primary ( >>> ) and secondary ( … ) prompts. The primary prompt is a way for the interpreter to let you know that it is expecting the next Python statement while the secondary prompt indicates that the interpreter is waiting for additional input to complete the current statement.

## Program Output, the `print` Statement, and "Hello World!"

Veterans to software development will no doubt be ready to take a look at the famous "Hello World!" program, typically the first program that a programmer experiences when exposed to a new language. There is no exception here.

```
>>> print 'Hello World!'
Hello World!
```

The `print` statement is used to display output to the screen. Those of you who are familiar with C are aware that the `printf()` function produces screen output. Many shell script languages use the `echo` command for program output.

**NOTE**

Usually when you want to see the contents of a variable, you use the `print` statement in your code. However, from within the interactive interpreter, you can use the `print` statement to give you the string representation of a variable, or just dump the variable raw—this is accomplished by simply giving the name of the variable.

In the following example, we assign a string variable, then use `print` to display its contents. Following that, we issue just the variable name.

```
>>> myString = 'Hello World!'
>>> print myString
Hello World!
>>> myString
'Hello World!'
```

Notice how just giving only the name reveals quotation marks around the string. The reason for this is to allow objects other than strings to be displayed in the same manner as this string —being able to display a printable string representation of any object, not just strings. The quotes are there to indicate that the object whose value you just dumped to the display is a string.

One final introductory note: The **print** statement, paired with the string format operator ( `%` ), behaves even more like C's `printf()` function:

```
>>> print "%s is number %d!" % ("Python", 1)
```

See Section 6.4 for more information on the string format and other operators.

## Program Input and the `raw_input()` Built-in Function

The easiest way to obtain user input from the command-line is with the `raw_input()` built-in function. It reads from standard input and assigns the string value to the variable you designate. You can use the `int()` built-in function (Python versions older than 1.5 will have to use the `string.atoi()` function) to convert any numeric input string to an integer representation.

```
>>> user = raw_input('Enter login name: ')
Enter login name: root
>>> print 'Your login is:', user
Your login is: root
```

The above example was strictly for text input. A numeric string input (with conversion to a real integer) example follows below:

```
>>> num = raw_input('Now enter a number: ')
Now enter a number: 1024
>>> print 'Doubling your number: %d' % (int(num) * 2)
Doubling your number: 2048
```

The `int()` function converts the string num to an integer, which is the reason why we need to use the `%d` (indicates integer) with the string format operator. See Section 6.5.3 for more information in the `raw_input()` built-in function.

## Comments

As with most scripting and Unix-shell languages, the hash/pound ( # ) sign signals that a comment begins right from the # and continues till the end of the line.

```
>>> # one comment
>>> print 'Hello World!'   # another comment
Hello World!
```

## Operators

The standard mathematical operators that you are familiar with work the same way in Python as in most other languages.

```
+          -          *          /          %          **
```

Addition, subtraction, multiplication, division, and modulus/remainder are all part of the standard set of operators. In addition, Python provides an exponentiation operator, the double star/asterisk ( `**` ). Although we are emphasizing the mathematical nature of these operators, please note that some of these operators are overloaded for use with other data types as well, for example, strings and lists.

```
>>> print -2 * 4 + 3 ** 2
1
```

As you can see from above, all operator priorities are what you expect: `+` and `-` at the bottom, followed by `*, /,` and `%,` then comes the unary `+` and `-,` and finally, `**` at the top. (`3 ** 2` is calculated first, followed by `-2 * 4,` then both results are summed together.)

**NOTE**

Although the example in the **`print`** statement is a valid mathematical statement, with Python's hierarchical rules dictating the order in which operations are applied, adhering to good programming style means properly placing parentheses to indicate visually the

grouping you have intended (see exercises). Anyone maintaining your code will thank you, and you will thank you.

Python also provides the standard comparison operators:

```
<           <=        >          >=         ==        !=    <>
```

Trying out some of the comparison operators we get:

```
>>> 2 < 4
1
>>> 2 == 4
0
>>> 2 > 4
0
>>> 6.2 <= 6
0
>>> 6.2 <= 6.2
1
>>> 6.2 <= 6.20001
1
```

Python currently supports two "not equals" comparison operators, `!=` and `<>`. These are the C-style and ABC/Pascal-style notations. The latter is slowly being phased out, so we recommend against its use.

Python also provides the expression conjunction operators:

```
and
                                                    or
                                                    not
```

Using these operators along with grouping parentheses, we can "chain" some of our comparisons together:

```
>>> (2 < 4) and (2 == 4)
0
>>> (2 > 4) or (2 < 4)
1
>>> not (6.2 <= 6)
```

```
1
>>> 3 < 4 < 5
```

The last example is an expression that maybe invalid in other languages, but in Python it is really a short way of saying:

```
>>> (3 < 4) and (4 < 5)
```

You can find out more about Python operators in Section 4.5of the text.

## Variables and Assignment

Rules for variables in Python are the same as they are in most other high-level languages: They are simply identifier names with an alphabetic first character—"alphabetic" meaning upper- or lowercase letters, including the underscore ( _ ). Any additional characters may be alphanumeric or underscore. Python is case-sensitive, meaning that the identifier "cAsE" is different from "CaSe."

Python is dynamically-typed, meaning that no pre-declaration of a variable or its type is necessary. The type (and value) are initialized on assignment. Assignments are performed using the equals sign.

```
>>> counter = 0
>>> miles = 1000.0
>>> name = 'Bob'
>>> counter = counter + 1
>>> kilometers = 1.609 * miles
>>> print '%f miles is the same as %f km' % (miles, kilometers)
1000.000000 miles is the same as 1609.000000 km
```

We have presented five examples of variable assignment. The first is an integer assignment followed by one each for floating point numbers, one for strings, an increment statement for integers, and finally, a floating point operation and assignment.

As you will discover in Chapter 3, the equals sign ( = ) was formerly the sole assignment operator in Python. However, beginning with 2.0, the equals sign can be combined with an arithmetic operation and the resulting value reassigned to the existing variable. Known as *augmented assignment,* statements such as:

```
n = n * 10
```

can now be written as:

```
n *= 10
```

Python does not support operators such as `n++` or `--n`.

The **`print`** statement at the end shows off the string format operator ( `%` ) again. Each "`%x`" code matches the type of the argument to be printed. We have seen `%s` (for strings) and `%d` (for integers) earlier in this chapter. Now we are introduced to `%f` (for floating point values). See Section 6.4 for more information on the string format operator.

## Numbers

Python supports four different numerical types:

**int** (signed integers)

**long** (long integers [can also be represented in octal and hexadecimal])

**float** (floating point real values)

**complex** (complex numbers)

Here are some examples:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *int* | 0101 | 84 | -237 | 0x80 | 017 | -680 | -0X92 |
| *long* | 29979062458L | -84140l | 0xDECADEDEADBEEFBADFEEDDEAL | | | | |
| *float* | 3.14159 | | 4.2E-10 | | -90. | 6.022e23 | -1.609E-19 |
| *complex* | 6.23+1.5j | | -1.23-875J | 0+1j | 9.80665-8.31441J | -.0224+0j | |

Numeric types of interest are the Python long and complex types. Python long integers should not be confused with C **`long`** s. Python longs have a capacity that surpasses any C **`long`**. You are limited only by the amount of (virtual) memory in your system as far as range is concerned. If you are familiar with Java, a Python long is similar to numbers of the `BigInteger` class type.

Complex numbers (numbers which involve the square root of -1, so called "imaginary" numbers) are not supported in many languages and perhaps are implemented only as classes in others.

All numeric types are covered in Chapter 5.

## Strings

Strings in Python are identified as a contiguous set of characters in between quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ( [ ] and [ : ] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end. The plus ( + ) sign is the string concatenation operator, and the asterisk ( * ) is the repetition operator. Here are some examples of strings and string usage:

```
>>> pystr = 'Python'
>>> iscool = 'is cool!'
>>> pystr[0]
'P'
>>> pystr[2:5]
'tho'
>>> iscool[:2]
'is'
>>> iscool[3:]
'cool!'
>>> iscool[-1]
'!'
>>> pystr + iscool
'Pythonis cool!'
>>> pystr + ' ' + iscool
'Python is cool!'
>>> pystr * 2
'PythonPython'
>>> '-' * 20
'--------------------'
```

You can learn more about strings in Chapter 6.

## Lists and Tuples

Lists and tuples can be thought of as generic "buckets" with which to hold an arbitrary number of arbitrary Python objects. The items are ordered and accessed via index offsets, similar to arrays, except that lists and tuples can store different types of objects.

The main differences between lists and tuples are: Lists are enclosed in brackets ( [ ] ), and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of for now as "read-only" lists. Subsets can be taken with the slice operator ( [] and [ : ] ) in the same manner as strings.

```
>>> aList = [1, 2, 3, 4]
>>> aList
```

```
[1, 2, 3, 4]
>>> aList[0]
1
>>> aList[2:]
[3, 4]
>>> aList[:3]
[1, 2, 3]
>>> aList[1] = 5
>>> aLlist
[1, 5, 3, 4]
```

Slice access to a tuple is similar, except for being able to set a value (as in `aList[1] = 5` above).

```
>>> aTuple = ('robots', 77, 93, 'try')
>>> aTuple
('robots', 77, 93, 'try')
>>> aTuple[0]
'robots'
>>> aTuple[2:]
(93, 'try')
>>> aTuple[:3]
('robots', 77, 93)
>>> aTuple[1] = 5
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn\qt support item assignment
```

We encountered an error in our last example because we attempted to update a tuple, which is not allowed. You can find out a lot more about lists and tuples along with strings in Chapter 6.

## Dictionaries

Dictionaries are Python's hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. Keys can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. Dictionaries are enclosed by curly braces ( { } ).

```
>>> aDict = {}
>>> aDict['host'] = 'earth'
>>> aDict['port'] = 80
>>> aDict
{'host': 'earth', 'port': 80}
>>> aDict.keys()
['host', 'port']
>>> aDict['host']
'earth'
```

Dictionaries are covered in Chapter 7.

## Code Blocks Use Indentation

Code blocks are identified by indentation rather than using symbols like curly braces. Without extra symbols, programs are easier to read. Also, indentation clearly identifies which block of code a statement belongs to. Of course, code blocks can consist of single statements, too.

When one is new to Python, indentation may comes as a surprise. Humans generally prefer to avoid change, so perhaps after many years of coding with brace delimitation, the first impression of using pure indentation may not be completely positive. However, recall that two of Python's features are that it is simplistic in nature and easy-to-read. Three hundred and sixty-five days after you indent your first line of Python, revisit this thought and determine if you maintain the same position you have today. More than likely, you will have discovered that life without braces is not as bad as you had originally thought.

## `if` Statement

The standard `if` conditional statement follows this syntax:

```
if expression:
    if_suite
```

If the *expression* is non-zero or true, then the statement *suite* is executed; otherwise, execution continues on the first statement after. "Suite" is the term used in Python to refer to a sub-block of code and can consist of single or multiple statements.

```
>>> if counter > 5:
…       print 'stopping after 5 iterations'
…       break
```

Python supports an `else` statement that is used with `if` in the following manner:

```
if expression:
     if_suite
else:
     else_suite
```

Python has an "else-if" statement named **elif** which has the following syntax:

```
if    expression1:
      if_suite
elif  expression2:
      elif_suite
else:
      else_suite
```

Another surprise: There is no `switch` or `case` statement in Python. This may also seem strange and/or detracting at first, but a set of **if-elif-else** statements are not as "ugly" because of Python's clean syntax. If you really want to circumvent a set of chained **if-elif-else** statements, another elegant workaround is using a **for** loop (see below) to iterate through your list of possible "cases."

You can learn more about **if, elif,** and **else** statements in the conditional section of Chapter 8.

## while Loop

The standard **while** conditional loop statement is similar to the **if.** Again, as with every code sub-block, indentation (and dedentation) are used to delimit blocks of code as well as to indicate which block of code statements belong to:

```
while  expression:
      while_suite
```

The statement *suite* is executed continuously in a loop until the expression becomes zero or false; execution then continues on the first succeeding statement.

```
>>> counter = 0
>>> while counter < 5:
…       print 'loop #%d' % (counter)
…       counter = counter + 1

loop #0
loop #1
loop #2
loop #3
loop #4
loop #5
```

Loops such as **while** and **for** (see below) are covered in the loops section of Chapter 8.

## for Loop and the range() Built-in Function

The **for** loop in Python is more like a foreach iterative-type loop in a shell scripting language than a traditional **for** conditional loop that works like a counter. Python's **for** loop takes what we will later describe as a sequence type (list, tuple, or string) and iterates over each element of that sequence.

```
>>> print 'I like to use the Internet for:'
I like to use the Internet for:
>>> for item in ['e-mail', 'net-surfing', 'homework', 'chat']:
…       print item
…
e-mail
net-surfing
homework
chat
```

Our output in the previous example may look more presentable if we display the items on the same line rather than on separate lines. **print** statements by default automatically add a NEWLINE character at the end of every line. This can be suppressed by terminating the **print** statement with a comma ( , ).

```
print 'I like to use the Internet for:'
for item in ['e-mail', 'net-surfing', 'homework', 'chat']:
     print item,
print
```

The code required further modification to include an additional **print** statement with no arguments to flush our line of output with a terminating NEWLINE; otherwise, the prompt will show up on the same line immediately after the last piece of data output. Here is the output with the modified code:

```
I like to use the Internet for:
e-mail net-surfing homework chat
```

Elements in **print** statements separated by commas will automatically include a delimiting space between them as they are displayed. Providing a string format gives the programmer the most control because it dictates the exact output layout, without having

to worry about the spaces generated by commas. It also allows all the data to be grouped together in one place—the tuple or dictionary on the right-hand side of the format operator.

```
>>> who = 'knights'
>>> what = 'Ni!'
>>> print 'We are the', who, 'who say', what, what, what, what
We are the knights who say Ni! Ni! Ni! Ni!
>>> print 'We are the %s who say %s' % \
…         who, ((what + ' ') * 4))
We are the knights who say Ni! Ni! Ni! Ni!
```

Using the string format operator also allows us to do some quick string manipulation before the output, as you can see in the above example.

We conclude our introduction to loops by showing you how we can make Python's **`for`** statement act more like a traditional loop, in other words, a numerical counting loop. Because we cannot change the behavior of a **`for`** loop (iterates over a sequence), we can manipulate our sequence so that it is a list of numbers. That way, even though we are still iterating over a sequence, it will at least appear to perform the number counting an incrementing that we envisioned.

```
>>> for eachNum in [0, 1, 2, 3, 4, 5]:
…         print eachNum
…
0
1
2
3
4
5
```

Within our loop, `eachNum` contains the integer value that we are displaying and can use it in any numerical calculation we wish. Because our range of numbers may differ, Python provides the `range()` built-in function to generate such a list for us. It does exactly what we want, taking a range of numbers and generating a list.

```
>>> for eachNum in range(6):
…         print eachNum
…
0
1
2
3
4
5
```

## Files and the open() Built-in Function

File access is one of the more important aspects of a language once you are comfortable with the syntax; there is nothing like the power of persistent storage to get some real work done.

### How to Open a File

```
handle = open( file_name, access_mode='r')
```

The *file_name* variable contains the string name of the file we wish to open, and *access_mode* is either `'r'` for read, `'w'` for write, or `'a'` for append. Other flags which can be used in the *access_mode* string include the `'+'` for dual read-write access and the `'b'` for binary access. If the mode is not provided, a default of read-only (`'r'`) is used to open the file.

**NOTE**

Attributes are items associated with a piece of data. Attributes can be simply data values or executable objects such as functions and methods. What kind of objects have attributes? Many. Classes, modules, files, complex numbers. These are just some of the Python objects which have attributes.

How do I access object attributes? With the dotted attribute notation, that is, by putting together the object and attribute names, separated by a dot or period: *object.attribute.*

If `open()` is successful, a file object will be returned as the handle (*handle*). All succeeding access to this file must go through its file handle. Once a file object is returned, we then have access to the other functionality through its methods such as `readlines()` and `close()`. Methods are *attributes* of file objects and must be accessed via the dotted attribute notation (see Core Note above).

Here is some code which prompts the user for the name of a text file, then opens the file and displays its contents to the screen:

```
filename = raw_input('Enter file name: ')
file = open(filename, 'r')
allLines = file.readlines()
```

```
file.close()
for eachLine in allLines:
   print eachLine,
```

Rather than looping to read and display one line at a time, our code does something a little different. We read all lines in one fell swoop, close the file, and *then* iterate through the lines of the file. One advantage to coding this way is that it permits the file access to complete more quickly. The output and file access do not have to alternate back and forth between reading a line and printing a line. It is cleaner and separates two somewhat unrelated tasks. The caveat here is the file size. The code above is reasonable for files with reasonable sizes. Programs too large may take up too much memory, in which case you would have to revert back to reading one line at a time.

The other interesting statement in our code is that we are again using the comma at the end of the **`print`** statement to suppress the printing of the NEWLINE character. Why? Because each text line of the file already contains NEWLINEs at the end of every line. If we did not suppress the NEWLINE from being added by **`print,`** our display would be double-spaced.

In Chapter 9, we cover file objects, their built-in methods attributes, and how to access your local file system. Please go there for all the details.

## Errors and Exceptions

Syntax errors are detected on compilation, but Python also allows for the detection of errors during program execution. When an error is detected, the Python interpreter *raises* (a.k.a. throws, generates, triggers) an exception. Armed with the information that Python's exception reporting can generate at runtime, programmers can quickly debug their applications as well as fine-tune their software to take a specific course of action if an anticipated error occurs.

To add error detection or exception handling to your code, just "wrap" it with a **`try-except`** statement. The suite following the **`try`** statement will be the code you want to manage. The code which comes after the **`except`** will be the code that executes if the exception you are anticipating occurs:

```
try:
     try_running_this_suite
except                                                  someError:
     suite_if_someError_occurs
```

Programmers can explicitly raise an exception with the **`raise`** command. You can learn more about exceptions as well as see a complete list of Python exceptions in Chapter 10.

# Functions

Functions in Python follow rules and syntax similar to most other languages: Functions are called using the functional operator ( ( ) ), functions must be declared before they are used, and the function type is the type of the value returned.

All arguments of function calls are made by reference, meaning that any changes to these parameters within the function affect the original objects in the calling function.

## How to Declare a Function

```
def
      function_name([ arguments]):
   "optional documentation string"
   function_suite
```

The syntax for declaring a function consists of the **def** keyword followed by the function name and any arguments which the function may take. Function arguments such as *arguments* above are optional, hence the reason why they are enclosed in brackets above. (Do not physically put brackets in your code!) The statement terminates with a colon (the same way that an **if** or **while** statement is terminated), and a code suite representing the function body follows. Here is one short example:

```
def addMe2Me(x):
   'apply + operation to argument'
   return (x + x)
```

This function, presumably meaning "add me to me" takes an object, adds its current value to itself and returns the sum. While the results are fairly obvious with numerical arguments, we point out that the plus sign works for almost all types. In other words, most of the standard types support the + operator, whether it be numeric addition or sequence concatenation.

## How to Call Functions

```
>>> addMe2Me(4.25)
8.5
>>>
>>> addMe2Me(10)
20
>>>
>>> addMe2Me('Python')
'PythonPython'
```

```
>>>
>>> addMe2Me([-1, 'abc'])
[-1, 'abc', -1, 'abc']
```

Calling functions in Python is similar to function invocations in other high-level languages, by giving the name of the function followed by the functional operator, a pair of parentheses. Any optional parameters go between the parentheses. Observe how the `+` operator works with non-numeric types.

### Default arguments

Functions may have arguments which have default values. If present, arguments will take on the appearance of assignment in the function declaration, but in actuality, it is just the syntax for default arguments and indicates that if a value is not provided for the parameter, it will take on the assigned value as a default.

```
>>> def foo(debug=1):
…       'determine if in debug mode with default argument'
…       if debug:
…           print 'in debug mode'
…       print 'done'
…
>>> foo()
in debug mode
done
>>> foo(0)
done
```

In the example above, the debug parameter has a default value of 1. When we do not pass in an argument to the function `foo()`, debug automatically takes on a true value of 1. On our second call to `foo()`, we deliberately send an argument of 0, so that the default argument is not used.

Functions have many more features than we could describe in this introductory section. Please refer to Chapter 11 for more details.

## Classes

A class is merely a container for static data members or function declarations, called a class's *attributes.* Classes provide something which can be considered a blueprint for creating "real" objects, called *class instances.* Functions which are part of classes are called methods. Classes are an object-oriented construct that are not required at this stage in learning Python. However, we will present it here for those who have some background in object-oriented methodology and would like to see how classes are implemented in Python.

## How to Declare a Class

```
class class_name[( base_classes_if_any)]:
   "optional documentation string"
   static_member_declarations
   method_declarations
```

Classes are declared using the **class** keyword. If a subclass is being declared, then the super or base classes from which it is derived is given in parentheses. This header line is then terminated and followed by an optional class documentation string, static member declarations, and any method declarations.

```
class FooClass:
    'my very first class: FooClass'
    version = 0.1            # class (data) attribute

    def __init__(self, nm='John Doe'):
        'constructor'
        self.name = nm       # class instance (data) attribute
        print 'Created a class instance for', nm

    def showname(self):
        'display instance attribute and class name'
        print 'Your name is', self.name
        print 'My name is', self.__class__    # full class name

    def showver(self):
        'display class(static) attribute'
        print self.version  # references FooClass.version

    def addMe2Me(self, x):  # does not use 'self'
        'apply + operation to argument'
        return (x + x)
```

In the above class, we declared one static data type variable `version` shared among all instances and four methods, `__init__()`, `showname()`, `showver()`, and the familiar `addMe2Me()`. The `show*()` methods do not really do much but output the data they were created to. The `__init__()` method has a special name, as do all those whose name begins and ends with a double underscore `( __ )`.

The `__init__()` method is a function provided by default that is called when a class instance is created, similar to a constructor and called after the object has been instantiated. Its purpose is to perform any other type of "start up" necessary for the instance to take on a life of its own. By creating our own `__init__()` method, we override the default method (which does not do anything) so that we can do customization and other "extra things" when our instance is created. In our case, we

initialize a class instance attribute called `name`. This variable is associated only with class instances and is not part of the actual class itself. `__init__()` also features a default argument, introduced in the previous section. You will no doubt also notice the one argument which is part of every method, `self`.

What is `self`? Self is basically an instance's handle to itself. (In other object-oriented languages such as C++ or Java, `self` is called `this`.) When a method is called, `self` refers to the instance which made the call. No class methods may be called without an instance, and is one reason why `self` is required. Class methods which belong to an instance are called *bound methods*. (Those not belonging to a class instance are called *unbound methods* and cannot be invoked [unless an instance is explicitly passed in as the first argument].)

## How to Create Class Instances

```
>>> foo1 = FooClass()
Created a class instance for John Doe
```

The string that is displayed is a result of a call to the `__init__()` method which we did not explicitly have to make. When an instance is created, `__init__()` is automatically called, whether we provided our own or the interpreter used the default one.

Creating instances looks just like calling a function and has the exact syntax. Class instantiation apparently uses the same functional operator as invoking a function or method. Do not get confused between the two, however. Just because the same symbols are used does not necessarily mean equivalent operations. Function calls and creating class instances are very different animals. The same applies for the `+` operator. Given a pair of integers, it performs integer addition; given a pair of floating point numbers, it performs real number addition; and giving it two strings results in string concatenation. All three of these operations are distinct.

Now that we have successfully created our first class instance, we can make some method calls, too:

```
>>> foo1.showname()
Your name is John Doe
My name is __main__.FooClass
>>>
>>> foo1.showver()
0.1
>>> print foo1.addMe2Me(5)
10
>>> print foo1.addMe2Me('xyz')
xyzxyz
```

The result of each function call is as we expected. One interesting piece of data is the class name. In the `showname()` method, we displayed the `self.__class__` variable which, for an instance, represents the name of the class from which it has been instantiated. In our example, we did not pass in a name to create our instance, so the `'John Doe'` default argument was used. In our next example, we do not use it.

```
>>> foo2 = FooClass('Jane Smith')
Created a class instance for Jane Smith
>>> foo2.showname()
Your name is Jane Smith
__main__.FooClass
```

There is plenty more on Python classes and instances in Chapter 13.

## Modules

Modules are a logical way to physically organize and distinguish related pieces of Python code into individual files. Modules can contain executable code, functions, classes, or any and all of the above.

When you create a Python source file, the name of the module is the same as the file except without the trailing "`.py`" extension. Once a module is created, you may "import" that module for use from another module using the **`import`** statement.

### How to Import a Module

```
import
                                                    module_name
```

### How to Call a Module Function or Access a Module Variable

Once imported, a module's attributes (functions and variables) can be accessed using the familiar dotted attribute notation:

```
module.function()
module.variable
```

We will now present our Hello World! example again, but using the output functions inside the `sys` module.

```
>>> import sys
>>> sys.stdout.write('Hello World!\n')
Hello World!
```

This code behaves just like our original Hello World! using the **print** statement. The only difference is that the standard output `write()` method is called, and the NEWLINE character needs to be stated explicitly because, unlike the **print** statement, `write()` does not do that for you.

Let us now look at some other attributes of the `sys` module and some of the functions in the `string` module as well.

```
>>> import sys
>>> import string
>>> sys.platform
'win32'
>>> sys.version
'1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)]'
>>>
>>> up2space = string.find(sys.version, ' ')
>>> ver = sys.version[:up2space]
>>> ver
1.5.2
>>>
>>> print 'I am running Python %s on %s' % (ver, sys.platform)
I am running Python 1.5.2 on win32
```

As you can probably surmise, the `sys.platform` and `sys.version` variables contain information regarding the system you are running on and which version of the Python interpreter you are using.

The `string.find()` function looks for substrings in strings. In the example, we are capturing just the version number which occurs from the beginning of the string to right before the first space character. We use `find()` to tell us where the space is located so we can grab all the characters in the string before the space.

Another way to snare the version number is by breaking up the entire string into words (separated by spaces). The version number is the first word, so that is all we want. The `string.split()` function returns a list of all the "words" in a string:

```
>>> verchunks = string.split(sys.version)
>>> verchunks
['1.5.2', '(#0,', 'Apr', '13', '1999,', '10:51:12)',
'[MSC', '32', 'bit', '(Intel)]']
>>> print 'I am running Python %s on %s' % \
…             (verchunks[0], sys.platform)
```

```
I am running Python 1.5.2 on win32
```

Our output is the exact same as the example above In this case, there was clearly more than one way to accomplish the same task. This is not always the case in Python, but both examples will allow the reader to decide on the appropriate course of action when warranted.

You can find out more information on modules and importing in Chapter 12.

We will cover all of the above topics in much greater detail throughout the text, but hopefully we have provided enough of a "quick dip in the pool" to facilitate your needs if your primary goal is to get started working with Python as quickly as possible without too much serious reading.

## Exercises

**1:**

*Variables, `print`, and the string format operator.* Start the interactive interpreter. Assign values to some variables (strings, numbers, etc.) and display them within the interpreter by typing their names. Also try doing the same thing with the `print` statement. What is the difference between giving just a variable name versus using it in conjunction with `print`? Also try using the string format operator ( `%` ) to become familiar with it.

**2:**

*Program output.* Take a look at the following Python script:

```
#!/usr/bin/env python
1 + 2 * 4
```

(a) What do you think this script does?

(b) What do you think this script will output?

(c) Type the code in as a script program and execute it. Did it do what you expected? Why or why not?

(d) How does execution differ if you are running this code from within the interactive interpreter? Try it and write down the results.

(e) How can we improve the output of the script version so that it does what we

expect/want?

**3:**

*Numbers and operators.* Enter the interpreter. Use Python to add, subtract, multiply, and divide two numbers (of any type). Then use the modulus operator to determine the remainder when dividing one number by another, and finally, raise one number to the power of another by using the exponentiation operator.

**4:**

*User input with* `raw_input()`.

(a) Create a small script to use `raw_input()` built-in function to take a string input from the user, then display to the user what he/she just typed in.

(b) Add another piece of similar code, but have the input be numeric. Convert the value to a number (using either `int()` or any of the other numeric conversion functions), and display the value back to the user. (Note that if your version of Python is older than 1.5, you will need to use the `string.ato*()` functions to perform the conversion.)

**5:**

*Loops and numbers.* Create some loops using both **`while`** and **`for`**.

(a) Write a loop that counts from 0 to 10 using a **`while`** loop. (Make sure your solution really *does* count from 0 to 10, not 0 to 9 or 1 to 10.)

(b) Do the same loop as in part (a), but use a **`for`** loop and the `range()` built-in function.

**6:**

*Conditionals.* Detect whether a number is positive, negative, or zero. Try using fixed values at first, then update your program to accept numeric input from the user.

**7:**

*Loops and strings.* Take a user input string and display string, one character at a time. As in your above solution, perform this task with a **`while`** loop first, then with a **`for`** loop.

**8:**

*Loops and operators.* Create a fixed list or tuple of 5 numbers and output their sum. Then update your program so that this set of numbers comes from user input. As with the problems above, implement your solution twice, once using

`while` and again with `for`.

**9:**

*More loops and operators.* Create a fixed list or tuple of 5 numbers and determine their average. The most difficult part of this exercise is the division to obtain the average. You will discover that integer division truncates and that you must use floating point division to obtain a more accurate result. The `float()` built-in function may help you there.

**10:**

*User input with loops and conditionals.* Use `raw_input()` to prompt for a number between 1 and 100. If the input matches criteria, indicate so on the screen and exit. Otherwise, display an error and reprompt the user until the correct input is received.

**11:**

*Menu-driven text applications.* Take your solutions to any number of the previous 5 problems and upgrade your program to present a menu-driven text-based application that presents the user with a set of choices, e.g., (1) sum of 5 numbers, (2) average of 5 numbers, … (X) Quit. The user makes a selection, which is then executed. The program exits when the user choose the "quit" option. The great advantage to a program like this is that it allows the user to run as many iterations of your solutions without having to necessarily restart the same program over and over again. (It is also good for the developer who is usually the first user main quality assurance engineer of their applications!)

**12:**

*The `dir()` built-in function.*

(a) Start up the Python interpreter. Run the `dir()` built-in function by simply typing "`dir()`" at the prompt. What do you see? Print the value of each element in the list you see. Write down the output for each and what you think each is.

(b) You may be asking, so what does `dir()` do? We have already seen that adding the pair of parentheses after "`dir`" causes the function to run. Try typing just the name "`dir`" at the prompt. What information does the interpreter give you? What do you think it means?

(c) The `type()` built-in function takes any Python object and returns its type. Try running it on dir by entering "`type(dir)`" into the interpreter. What do you get?

(d) For the final part of this exercise, let us take a quick look at Python documentation strings. We can access the documentation for the dir() function by

appending "`.__doc__`" after its name. So from the interpreter, display the document string for `dir()` by typing the following at the promt: **`print`** `dir.__doc__`. Many of the built-in functions, methods, modules, and module attributes have a documentation string associated with them. We invite you to put in your own as you write your code; it may help another user down the road.

**13:**

*Finding more about the `sys` module with `dir()`.*

(a) Start the Python interpreter again. Run the `dir()` command as in the previous exercise. Now import the `sys` module by typing **`import`** `sys` at the prompt. Run the `dir()` command again to verify that the sys module now shows up. Now run the `dir()` command on the `sys` module by typing `dir(sys)`. Now you see all the attributes of the sys module.

(b) Display the version and platform variables of the `sys` module. Be sure to prepend the names with `sys` to indicate that they are attributes of `sys`. The version variable contains information regarding the version of the Python interpreter you are using, and the platform attribute contains the name of the computer system that Python believes you are running on.

(c) Finally, call the `sys.exit()` function. This is another way to quit the Python interpreter in case the keystrokes described above in problem 1 do not get you out of Python.

**14:**

*Operator precedence and grouping with parentheses.* Rewrite the mathematical expression of the **`print`** statement in Section 2.4, but try to group pairs of operands correctly, using parentheses.

**15:**

Elementary sorting.

(a) Have the user enter 3 numeric values and store them in 3 different variables. Without using lists or sorting algorithms, manually sort these 3 numbers from smallest to largest.

(b) How would you change your solution in part (a) to sort from largest to smallest?

**16:**

*Files.* Type in and/or run the file display code in Section 2.14. Verify that it works

on your system and try different input files as well.

# Chapter 3. Syntax and Style

Our next goal is to go through the basic Python syntax, describe some general style guidelines, then be briefed on identifiers, variables, and keywords. We will also discuss how memory space for variables is allocated and deallocated. Finally, we will be exposed to a much larger example Python program—taking the plunge, as it were. No need to worry, there are plenty of life preservers around that allow for swimming rather than the alternative.

## Statements and Syntax

Some rules and certain symbols are used with regard to statements in Python:

Hash mark ( # ) indicates Python comments

NEWLINE ( \n ) is the standard line separator (one statement per line)

Backslash ( \ ) continues a line

Semicolon ( ; ) joins two statements on a line

Colon ( : ) separates a header line from its suite

Statements (code blocks) grouped as suites

Suites delimited via indentation

Python files organized as "modules"

### Comments ( # )

First thing's first: Although Python is one of the easiest languages to read, it does not preclude the programmer from proper and adequate usage and placement of comments in the code. Like many of its Unix scripting brethren, Python comment statements begin with the pound sign or hash symbol (#). A comment can begin anywhere on a line. All characters following the # to the end of the line are ignored by the interpreter. Use them wisely and judiciously.

### Continuation ( \ )

Python statements are, in general, delimited by NEWLINEs, meaning one statement per line. Single statements can be broken up into multiple lines by use of the backslash. The

backslash symbol ( \ ) can be placed before a NEWLINE to continue the current statement onto the next line.

```
# check conditions
if (weather_is_hot == 1) and \
   (shark_warnings == 0) :

    send_goto_beach_mesg_to_pager()
```

There are two exceptions where lines can be continued without backslashes. A single statement can take up more than one line when (1) container objects are broken up between elements over multiple lines, and when (2) NEWLINEs are contained in strings enclosed in triple quotes.

```
# display a string with triple quotes
print '''hi there, this is a long message for you
that goes over multiple lines… you will find
out soon that triple quotes in Python allows
this kind of fun! it is like a day on the beach!'''

# set some variables
go_surf, get_a_tan_while, boat_size, toll_money = ( 1,
   'windsurfing',  40.0,  -2.00 )
```

## Multiple Statement Groups as Suites ( : )

Groups of individual statements making up a single code block are called "suites" in Python (as we introduced in Chapter 2). *Compound* or *complex statements,* such as **if, while, def,** and **class,** are those which require a header line and a suite. Header lines begin the statement (with the keyword) and terminate with a colon ( : ) and are followed by one or more lines which make up the suite. We will refer to the combination of a header line and a suite as a *clause.*

## Suites Delimited via Indentation

As we introduced in Section 2.10, Python employs indentation as a means of delimiting blocks of code. Code at inner levels are indented via spaces or TABs. Indentation requires exact indentation, in other words, all the lines of code in a suite must be indented at the exact same level (e.g. same number of spaces). Indented lines starting at different positions or column numbers is not allowed; each line would be considered part of another suite and would more than likely result in syntax errors.

A new code block is recognized when the amount of indentation has increased, and its termination is signaled by a "dedentation," or a reduction of indentation matching a

previous level's. Code that is not indented, i.e., the highest level of code, is considered the "main" portion of the script.

The decision to creating Python using indentation was based on the belief that grouping code in this manner is more elegant and contributes to the ease-of-reading to which we alluded earlier. It also helps avoid "dangling-else"-type problems, including ungrouped single statement clauses (those where a C `if` statement which does not use braces at all, but has two indented statements following). The second statement will execute regardless of the conditional, leading to more programmer confusion until the light bulb finally blinks on.

Finally, no "holy brace wars" can occur when using indentation. In C (also C++ and Java), starting braces may be placed on the same line as the header statement, or may start the very next line, or may be indented on the next line. Some like it one way, others prefer the other, etc. You get the picture.

We should also mention a minor performance improvement which can occur since each missing brace means one less byte to load during execution. Sure these are pennies on their own, but add up to hundreds and thousands of bytes over a 24×7×365 environment across a global network such as the Internet and you have something you can see. See below in Section 3.4 for tips and style guidelines on indentation.

## Multiple Statements on a Single Line ( ; )

The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

We caution the reader to be wary of the amount of usage of chaining multiple statements on individual lines since it makes code much less readable. You decide:

```
import sys
x = 'foo'
sys.stdout.write(x + '\n')
```

In our example, separating the code to individual lines makes for remarkably improved reader-friendliness.

## Modules

Each Python script is considered a *module.* Modules have a physical presence as disk files. When a module gets large enough or has diverse enough functionality, it may make sense

to move some of the code out to another module. Code that resides in modules may belong to an application (i.e., a script that is directly executed), or may be executable code in a library-type module that may be "imported" from another module for invocation. As we mentioned in the last chapter, modules can contain blocks of code to run, class declarations, function declarations, or any combination of all of those.

## Variable Assignment

This section focuses on variable assignment. We will discuss which identifiers make valid variables coming up in Section 3.3.

### Equal sign ( = ) is the assignment operator

The equal sign ( = ) is the main Python assignment operator

```
anInt = -12
String  =  'cart'
aFloat = -3.1415 *  (5.0 ** 2)
anotherString = 'shop'  + 'ping'
aList = [ 3.14e10, '2nd  elmt of a list', 8.82-4.371j ]
```

Be aware now that assignment does not explicitly assign a value to a variable, although it may appear that way from your experience with other programming languages. In Python, objects are referenced, so on assignment, a reference (not a value) to an object is what is being assigned, whether the object was just created or was a pre-existing object. If this is not 100% clear now, do not worry about it. We will revisit this topic later on in the chapter, but just keep it in mind for now.

Also, if you familiar with C, you are aware that assignments are treated as expressions. This is not the case for Python, where assignments do not have inherent values. Statements such as the following are invalid in Python:

```
>>> x = 1
>>> y = (x = x + 1)  # assignments not expressions!
  File "<stdin>", line 1
    y = (x = x + 1)
           ^
SyntaxError: invalid syntax
```

Beginning in Python 2.0, the equals sign can be combined with an arithmetic operation and the resulting value reassigned to the existing variable. Known as *augmented assignment,* statements such as

```
x = x + 1
```

can now be written as

```
x += 1
```

Python does not support pre-/post-increment nor pre-/post-decrement operators such as `x++` or `--x`.

## How To Do a Multiple Assignment

```
>>> x = y = z = 1
>>> x
1
>>> y
1
>>> z
1
```

In the above example, an integer object (with the value 1) is created, and `x, y,` and `z` are all assigned the same reference to that object. This is the process of assigning a single object to multiple variables. It is also possible in Python to assign multiple objects to multiple variables.

## How to Do a "Multuple" Assignment

Another way of assigning multiple variables is using what we shall call the "multuple" assignment. This is not an official Python term, but we use "multuple" here because when assigning variables this way, the objects on both sides of the equals sign are tuples, a Python standard type we introduced in Section 2.8.

```
>>> x, y, z = 1, 2, 'a string'
>>> x
1
>>> y
2
>>> z
'a string'
```

In the above example, two integer objects (with values 1 and 2) and one string object are assigned to `x, y,` and `z` respectively. Parentheses are normally used to denote tuples, and

although they are optional, we recommend them anywhere they make the code easier to read:

```
>>> (x, y, z) = (1, 2, 'a string')
```

If you have ever needed to swap values in other languages like C, you will be reminded that a temporary variable, i.e., `tmp`, is required to hold one value which the other is being exchanged:

```
/* swapping variables in C */
tmp = x;
x = y;
y = tmp;
```

In the above C code fragment, the values of the variables `x` and `y` are being exchanged. The `tmp` variable is needed to hold the value of one of the variables while the other is being copied into it. After that step, the original value kept in the temporary variable can be assigned to the second variable.

One interesting side effect of Python's "multuple" assignment is that we no longer need a temporary variable to swap the values of two variables.

```
# swapping variables in Python
>>> (x, y) = (1, 2)
>>> x
1
>>> y
2
>>> (x, y) = (y, x)
>>> x
2
>>> y
1
```

Obviously, Python performs evaluation before making assignments.

## Identifiers

Identifiers are the set of valid strings which are allowed as names in a computer language. From this all-encompassing list, we segregate out those which are *keywords,* names that form a construct of the language. Such identifiers are reserved words which may not be used for any other purpose, or else a syntax error (`SyntaxError` exception) will occur.

Python also has an additional set of identifiers known as *built-ins,* and although they are not reserved words, use of these special names is not recommended. (Also see Section 3.3.3.)

## Valid Python Identifiers

The rules for Python identifier strings are not unlike most other high-level programming languages:

First character must be letter or underscore ( _ )

Any additional characters can be alphanumeric or underscore

Case-sensitive

No identifiers can begin with a number, and no symbols other than the underscore are ever allowed. The easiest way to deal with underscores is to consider them as alphabetic characters. *Case-sensitivity* means that identifier `foo` is different from `Foo,` and both of those are different from `FOO.`

## Keywords

Python currently has twenty-eight keywords. They are listed in Table 3.1.

Generally, the keywords in any language should remain relatively stable, but should things ever change (as Python is a growing and evolving language), a list of keywords as well as an `iskeyword()` function are available in the `keyword` module.

**Table 3.1. Python Keywords**

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

For compatibility reasons, observe that the **assert** keyword is new as of Python 1.5, and the **access** keyword was obsolete beginning with 1.4.

## Built-ins

In addition to keywords, Python has a set of "built-in" names which are either set and/or used by the interpreter that are available at any level of Python code. Although not keywords, built-ins should be treated as "reserved for the system" and not used for any other purpose. However, some circumstances may call for *overriding* (a.k.a. redefining,

replacing) them. Python does not support overloading of identifiers, so only one name "binding" may exist at any given time.

### Special Underscore Identifiers

Python designates (even more) special variables with underscores both prefixed and suffixed. We will also discover later that some are quite useful to the programmer while others are unknown or useless. Here is a summary of the special underscore usage in Python:

`_xxx` do not import with `'from module import *'`

`__xxx__` system-defined name

`__xxx` request private name mangling in classes

**NOTE**

Because of the underscore usage in Python system, interpreter, and built-in identifiers, we recommend that the programmer avoid the use of beginning variable names with the underscore.

## Basic Style Guidelines

### Comments

You do not need to be reminded that comments are useful both to you and those who come after you. This is especially true for code that has been untouched by man (or woman) for a time (that means several months in software development time). Comments should not be absent, nor should there be novellas. Keep the comments explanatory, clear, short, and concise, but get them in *there.* In the end, it saves time and energy for everyone.

### Documentation

Python also provides a mechanism whereby documentation strings can be retrieved dynamically through the `__doc__` special variable. The first unassigned string in a module, class declaration, or function declaration can be accessed through by using `obj.__doc__` where `obj` is the module, class, or function name.

### Indentation

Since indentation plays a major role, you will have to decide on a spacing style that is easy to read as well as the least confusing. Common sense also plays a recurring role in choosing how many spaces or columns to indent.

| | |
|---|---|
| 1 or | probably not enough; difficult to determine which block of code statements |

| | |
|---|---|
| 2 | belong to |
| 8 to 10 | may be too many; code which has many embedded levels will wraparound, causing the source to be difficult to read |

Four (4) spaces is very popular, not to mention being the preferred choice of Python's creator. Five (5) and six (6) are not bad, but text editors usually do *not* use these settings, so they are not as commonly used. Three (3) and seven (7) are borderline cases.

As far as TABs go, bear in mind that different text editors have different concepts of what TABs are. It is advised not to use TABs if your code will live and run on different systems or be accessed with different text editors.

## Choosing Identifier Names

The concept of good judgment also applies in choosing logical identifier names. Decide on short yet meaningful identifiers for variables. Although variable length is no longer an issue with programming languages of today, it is still a good idea to keep name sizes reasonable. The same applies for naming your modules (Python files).

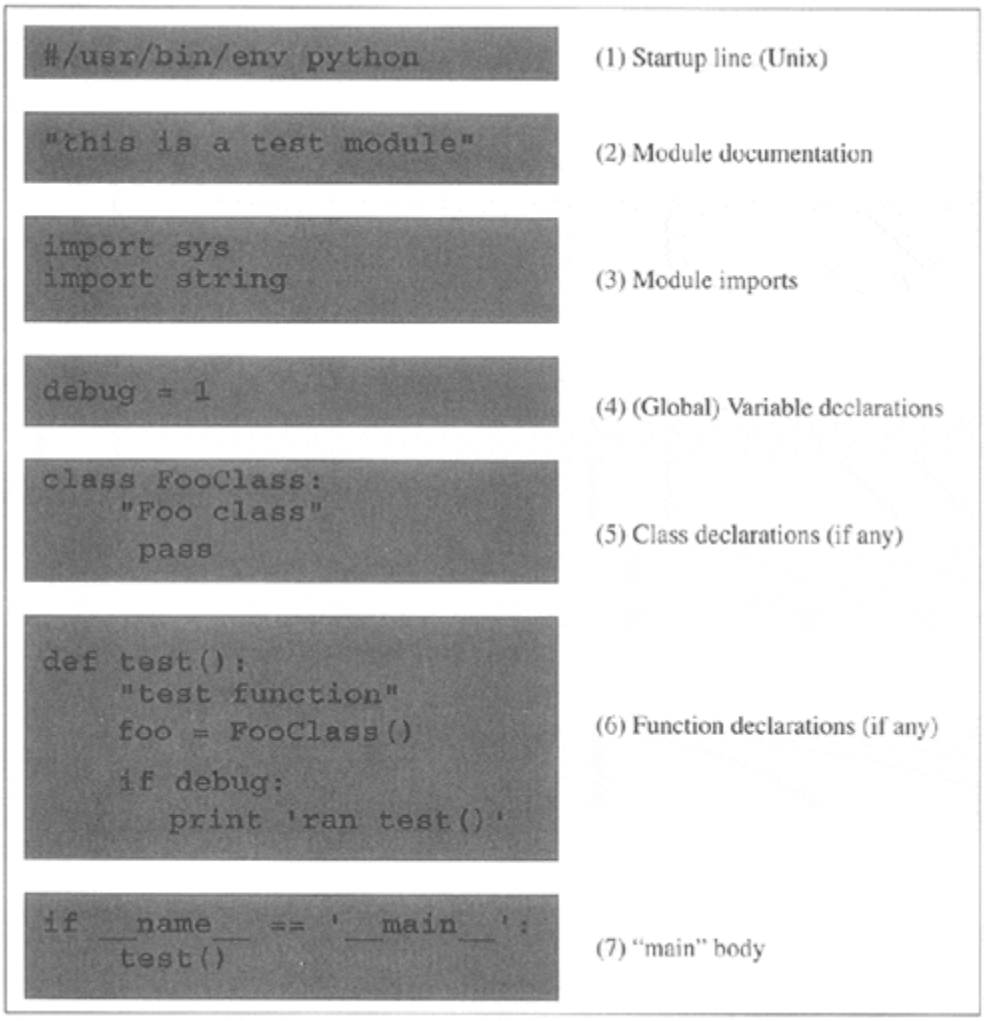
## Module Structure and Layout

Modules are simply physical ways of logically organizing all your Python code. Within each file, you should set up a consistent and easy-to-read structure. One such layout is the following:

```
# (1) startup line (Unix)
# (2) module documentation
# (3) module imports
# (4) variable declarations
# (5) class declarations
# (6) function declarations
# (7) "main" body
```

Figure 3-1 illustrates the internal structure of a typical module.

**Figure 3-1. Typical Python File Structure**




(1) Startup line

Generally used only in Unix environments, the start-up line allows for script execution by name only (invoking the interpreter is not required).

(2) Module documentation

Summary of a module's functionality and significant global variables; accessible externally as `module.__doc__`.

(3) Module imports

Import all the modules necessary for all the code in current module; modules are imported once (when this module is loaded); imports within functions are not invoked until those functions are called.

(4) Variable declarations

Declare (global) variables here which are used by multiple functions in this module (if not, make them local variables for improved memory/performance).

(5) Class declarations

Any classes should be declared here, along with any static member and method attributes; class is defined when this module is imported and the class statement executed. Documentation variable is `class.__doc__`.

(6) Function declarations

Functions which are declared here are accessible externally as `module.function();` function is defined when this module is imported and the **`def`** statement executed. Documentation variable is `function.__doc__`.

(7) "main" body

All code at this level is executed, whether this module is imported or started as a script; generally does not include much functional code; rather, gives direction depending on mode of execution.

**NOTE**

The main body of code tends to contain lines such as the ones you see above which check the **`__name__`** variable and takes appropriate action (see Core Note below). Code in the main body typically executes the class, function, and variable declarations, then checks **`__name__`** to see whether it should invoke another function (often called **`main()`**) which performs the primary duties of this module. The main body usually does no more than that. (Our example above uses **`test()`** rather than **`main()`** to avoid confusion until you read this Core Note.)

Regardless of the name, we want to emphasize that this is a great place to put a test suite in your code. As we explain in *Section 3.4.2*, most Python modules are created for import use only, and calling such a module directly should invoke a regression test of the code in such a module.

Most projects tend to consist of a single application and importing any required modules. Thus it is important to bear in mind that most modules are created solely to be imported rather than to execute as scripts. We are more likely to create a Python library-style module whose sole purpose is to be imported by another module. After all, only one of the modules—the one which houses the main application—will be executed, either by a user from the command-line, by a batch or timed mechanism such as a Unix cron job, via a web server call, or be invoked from a GUI callback.

With that fact in hand, we should also remember that all modules have the ability to execute code. All Python statements in the highest level of code, that is, the lines that are not indented, will be executed on import, whether desired or not. Because of this "feature," safer code is written such that everything is in a function except for the code that should be executed on an import of a module. Again, usually only the main application module has the bulk of the executable code at its highest-level. All other imported modules will have very little on the outside, and everything in functions or classes. (See Core Note below for more information.)

**NOTE**

Because the "main" code is executed whether a module is imported or executed directly, we often need to know how this module was loaded to guide the execution path. An application may wish to import the module of another application, perhaps to access useful code which will otherwise have to be duplicated (not the OO thing to do). However, in this case, you only want access to this other application's code, not to necessarily run it. So the big question is, "Is there a way for Python to detect at runtime whether this module was imported or executed directly?" The answer is… (drum roll…) yes! The `__name__` system variable is the ticket.

**`__name__`** contains module name if imported

**`__name__`** contains **`'__main__'`** if executed directly

## Create Tests in the Main Body

For good programmers and engineers, providing a test suite or harness for our entire application is the goal. Python simplifies this task particularly well for modules created solely for import. For these modules, you know that they would never be executed directly. Wouldn't it be nice if they were invoked to run code that puts that module through the test grinder? Would this be difficult to set up? Not really.

The test software should run only when this file is executed directly, i.e., not when it is imported from another module, which is the usual case. Above and in the Core Note, we described how we can determine whether a module was imported or executed directly. We can take advantage of this mechanism by using the `__name__` variable. If this module was called as a script, plug the test code right in there, perhaps as part of `main()` or `test()` (or whatever you decide to call your "second-level" piece of code) function, which is called only if this module is executed directly.

The "tester" application for our code should be kept current along with any new test criteria and results, and it should run as often as the code is updated. These steps will help improve the robustness of our code, not to mention validating and verifying any new features or updates.

## Memory Management

So far you have seen a large number of Python code samples and may have noticed a few interesting details about variables and memory management. Highlighting some of the more conspicuous ones, we have:

Variables not declared ahead of time

Variable types are not declared

No memory management on programmers' part

Variable names can be "recycled"

`del` statement allows for explicit "deallocation"

### Variable Declarations (or Lack Thereof)

In most compiled languages, variables must be declared before they are used. In fact, C is even more restrictive: Variables have to be declared at the beginning of a code block and before any statements are given. Other languages, like C++ and Java, allow "on-the-fly" declarations i.e., those which occur in the middle of a body of code—but these name and type declarations are still required before the variables can be used. In Python, there are no explicit variable declarations. Variables are "declared" on first assignment. Like most languages, however, variables cannot be accessed until they are (created and) assigned:

```
>>> a
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: a
```

Once a variable has been assigned, you can access it by using its name:

```
>>> x = 4
>>> y = 'this is a string'
>>> x
4
>>> y
'this is a string'
```

### Dynamic Typing

Another observation, in addition to lack of variable declaration, is the lack of type specification. In Python, the type and memory space for an object are determined and

allocated at run-time. Although code is byte-compiled, Python is still an interpreted language. On creation, that is, on assignment, the interpreter creates an object whose type is dictated by the syntax that is used for the operand on the right-hand side of an assignment. After the object is created, a reference to that object is assigned to the variable on the left-hand side of the assignment.

## Memory Allocation

As responsible programmers, we are aware that when allocating memory space for variables, we are borrowing system resources, and eventually, we will have to return that which we borrowed back to the system. Happily, not only do we not have to explicitly allocate the memory, we don't have to deallocate it either. That is memory management made easy. Well, okay, perhaps it had something to do with the decision that Python should simply be a tool for the application writer to and shouldn't have to worry about lower-level, operating system or machine-oriented tasks.

## Garbage Collection

Memory that is no longer being used is reclaimed by the system using a mechanism known as *garbage collection.* Python's garbage collector will automatically deallocate a data object once is it no longer needed, all without requiring any management on the programmer's part. How does Python decide when an object is "no longer needed?" By keeping track of the number of references to objects. This is called *reference counting.*

## Reference Counting

To keep track of memory that has been allocated, Python does something quite similar to card-counting, a popular scheme used in casino gaming. When an object is created, a reference is made to that object. An internal tracking variable, a reference counter, keeps track of how many references are being made to each object. The reference count for an object is initially set to one (1) when an object is created and (its reference) assigned.

New references to objects, also called *aliases,* occur when additional variables are assigned to the same object, passed as arguments to invoke other bodies of code such as functions, methods, or class instantiation, or assigned as members of a sequence or mapping.

```
# initialize string object, set reference count to 1
foo1 = 'foobar'

# increment reference count by assigning another variable
foo2 = foo1  # create an alias

# increment ref count again temporarily by calling function
check_val(foo1)
```

In the function call above, the reference count is set to one on creation, incremented when an alias is created, and incremented again when the object participated in a function call. The reference count is decremented when the function call has completed; and once again if `foo2` is removed from the namespace. The reference count goes to zero and the object deallocated when `foo1` goes out of scope. (See Section 11.8 for more information on variable scope.)

## `del` Statement

The `del` statement removes a single reference to an object, and its syntax is:

```
del obj1[, obj2[, … objN… ]]
```

For example, executing `del foo2` in the example above has two results:

(1) removes name `foo2` from namespace

(2) lowers reference count to object `'foobar'` (by one)

Further still, executing `del foo1` will remove the final reference to the `'foobar'` object, decrementing the reference counter to zero and causing the object to become "inaccessible" or "unreachable." It is at this point that the object becomes a candidate for garbage collection. Note that any tracing or debugging facility may keep additional references to an object, delaying or postponing that object from being garbage-collected.

## Decrementing Reference Count

You already noticed that when the `del` statement was executed, an object was not really "deleted," rather just a reference to it. Likewise, you can "lose" the reference to an object by reassigning it to another object.

```
foo1 = 'foobar'          # create original string
foo1 = 'a new string'    # 'foobar' "lost" and reclaimed
```

The preceding example shows how all references to an object can occur with reassigning a variable. The most common case utilizes neither reassignment nor calling the `del` statement.

Exiting from the current *scope* means that when a piece of code such as a function or method has completed, all the objects created within that scope are destroyed (unless passed back as a return object), such as our example above when `foo1` is given as an argument to the `check_val()` function. The reference count for `foo1` is incremented on the call and decremented when the function completed.

We present below a reference count decrementing summary. The reference count for an object is decremented when a variable referencing the object…

Is named explicitly in a **del** statement

Is (re)assigned to another object

Goes out-of-scope

## First Python Application

Now that we are familiar with the syntax, style, variable assignment and memory allocation, it is time to look at a more complex example of Python programming. Many of the things in this program will be parts of Python which may have unfamiliar constructs, but we believe that Python is so simple and elegant that the reader should be able to make the appropriate conclusions upon examination of the code.

The source file we will be looking at is `fgrepwc.py`, named in honor of the two Unix utilities of which this program is a hybrid. `fgrep` is a simple string searching command. It looks at a text file line by line and will output any line for which the search string appears. Note that a string may appear more than once on a line. `wc` is another Unix command; this one counts the number of characters, words, and lines of an input text file.

Our version does a little of both. It requires a search string and a filename, and outputs all lines with a match and concludes by displaying the total number of matching lines found. Because a string may appear more than once on a line, we have to state that the count is a strict number of lines that match rather than the total number of times a search string appears in a text file. (One of the exercises at the end of the chapter requires the reader to "upgrade" the program so that the output is the total number of matches.)

One other note before we take a look at the code: The normal convention for source code in this text is to leave out all comments, and place the annotated version on the CD-ROM. However, we will include comments for this example to aid you as you explore your first longer Python script with features we have yet to introduce.

We now introduce `fgrepwc.py`, found below as Listing 3.1, and provide analysis immediately afterward.

### Example 3.1. File Find (**fgrepwc.py**)

This application looks for a search word in a file and displays each matching line as well as a summary of how many matching lines were found.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  "fgrepwc.py -- searches for string in text file"
004 4
```

```
005 5  import sys
006 6  import string
007 7
008 8  # print usage and exit
009 9  def usage():
010 10     print "usage:  fgrepwc [ -i ] string file"
011 11     sys.exit(1)
012 12
013 13 # does all the work
014 14 def filefind(word, filename):
015 15
016 16     # reset word count
017 17     count = 0
018 18
019 19     # can we open file? if so, return file handle
020 20     try: <$nopage>
021 21         fh = open(filename, 'r') <$nopage>
022 22
023 23     # if not, exit
024 24     except: <$nopage>
025 25         print filename, ":",sys.exc_info()[1]
026 26         usage()
027 27
028 28     # read all file lines into list and close
029 29     allLines = fh.readlines()
030 30     fh.close()
031 31
032 32     # iterate over all lines of file
033 33     for eachLine in allLines:
034 34
035 35         # search each line for the word
036 36         if string.find(eachLine, word) > -1:
037 37             count = count + 1
038 38             print eachLine,
039 39
040 40     # when complete, display line count
041 41     print count
042 42
043 43 # validates arguments and calls filefind()
044 44 def checkargs():
045 45
046 46     # check args; 'argv' comes from 'sys' module
047 47     argc = len(sys.argv)
048 48     if argc != 3:
049 49         usage()
050 50
051 51     # call fgrepwc.filefind() with args
052 52     filefind(sys.argv[1], sys.argv[2])
053 53
054 54 # execute as application
055 55 if __name__ == '__main__':
056 56     checkargs()
057  <$nopage>
```

## Lines 1–3

The Unix start up line is followed by the module documentation string. If you import the `fgrepwc` module from another module, this string can be accessed with `fgrepwc.__doc__`. This is a key feature because it makes previously static text information available in a dynamic execution environment. We can also point out that what we described is usually the only use of the documentation string. It serves no other purpose, but it can double as a comment which is conveniently located at the top of a file. (We invite the reader to take a look at the documentation string at the commencement of the `cgi` module in the standard library for a serious example of module documentation.)

## Lines 5–6

We've already seen the `sys` and `string` modules. The `sys` module contains mostly variables and functions that represent interaction between the Python interpreter and the operating system. You will find items in here such as the command-line arguments, the `exit()` function, the contents of the Python path environment variable `PYTHONPATH`, the standard files, and information on errors.

The `string` module contains practically every function you'll need in processing strings, such as integer conversion via `atoi()` (and related functions), various string variables, and other string manipulation functions.

The main motivation to provide modules to import is to keep the language small, light, fast, and efficient, and bring in only software that you need to get the job done. Plug'n'play with only the modules you need. Perl and Java have a similar setup, importing modules, packages, and the like, and to a certain extent so do C and C++ with the inclusion of header files.

## Lines 8–11

We declare a function called `usage()` here which has no arguments/parameters. The purpose of this function is to simply display a message to the user indicating the proper command-line syntax with which to initiate the script, and exit the program with the `exit()` function, found in the `sys` module. We also mentioned that in the Python namespace, calling a function from an imported module requires a "fully-qualified" name. All imported variables and functions have the following formats: *`module`*`.variable` or *`module`*`.function()`. Thus we have `sys.exit()`.

An alternative **`from-import`** statement allows the import of specific functions or variables from a module, bringing them into the current namespace. If this method of importing is used, only the attribute name is necessary.

For example, if we wanted to import only the `exit()` function from `sys` and nothing else, we could use the following replacement:

```
from sys import exit
```

Then in the `usage()` function, we would call `exit(1)` and leave off the "`sys.`". One final note about `exit()`: The argument to `sys.exit()` is the same as the C `exit()` function, and that is the return value to the calling program, usually a command-line shell program. With that said, we point out that this "protocol" of printing usage and exiting applies only to command-line driven applications.

In web-based applications, this would not be the preferred way to quit a running program, because the calling web browser is expecting an acceptable valid HTML response. For web applications, it is more appropriate to output an error message formatted in HTML so that end-users can correct their input. So, basically, no web application should terminate with an error. Exiting a program will send a system or browser error to the user, which is incorrect behavior and the responsibility falls on the website application developer.

The same theory applies to GUI-based applications, which should not "crash out" of their executing window. The correct way to handle errors in such applications is to bring up an error dialog and notify the user and perhaps allow for a parameter change which may rectify the situation.

## Lines 13–41

The core part of our Python program is the `filefind()` function. `filefind()` takes two parameters: the word the user is searching for, and the name of the file to search.

A counter is kept to track the total number of successful matches (number of lines that contain the word). The next step is to open the file. The **`try-except`** construct is used to "catch" errors which may occur when attempting to open the file. One of Python's strengths is its ability to let the programmer handle errors and perform appropriate action rather than simply exiting the program. This results in a more robust application and a more acceptable way of programming. Chapter 10 is devoted to errors and exceptions.

Barring any errors, the goal of this section of function is to open a file, read in all the lines into a buffer that can be processed later, and close the file. We took a sneak peek at files earlier, but to recap, the `open()` built-in function returns a file object or file handle, with which all succeeding operations are performed on, i.e., `readlines()` and `close()`.

The final part of the function involves iterating through each line, looking for the target word. Searching is accomplished using the `find()` function from the string module. `find()` returns the starting character position (index) if there is a match, or -1 if the string does not appear in the line. All successful matches are tallied and matching lines are displayed to the user.

`filefind()` concludes by displaying the total number of matching lines that were found.

## Lines 43–52

The last function found in our program is `checkargs(),` which does exactly two things: checking for the correct number of command-line arguments and calling `filefind()` to do the real work. The command-line arguments are stored in the `sys.argv` list. The first argument is the program name and presumably, the second is the string we are looking for, and the final argument is the name of the file to search.

### Lines 54–56

This is the special code we alluded to earlier: the code that determines (based on `__name__`) the different courses of action to take if this script was imported or executed directly. With the boilerplate **`if`** statement, we can be sure that `checkargs()` would not be executed if this module were imported, nor would we want it to. It exits anyway because the check for the command-line arguments would fail. If the code did not have the **`if`** statement and the main body of code consisted of just the single line to call `checkargs(),` then `checkargs()` would be executed whether this module was imported or executed directly.

One final note regarding `fgrepwc.py.` This script was created to run from the command-line. Some work would be required, specifically interface changes, if you wanted to execute this from a GUI or web-based environment.

The example we just looked at was fairly complex, but hopefully it was not a complete mystery, with the help of our comments in this section as well as any previous programming experience you may have brought. In the next chapter, we will take a closer look at Python objects, the standard data types, and how we can classify them.

## Exercises

**1:**

*Identifiers.* Why are variable type declarations not used in Python?

**2:**

*Identifiers.* Why are variable name declarations not used in Python?

**3:**

*Identifiers.* Why should we avoid the use of the underscore to begin variable names with?

**4:**

*Statements.* Can multiple Python statements be written on a single line?

**5:**

*Statements.* Can a single Python statement be written over multiple lines?

**6:**

*Variable assignment.*

(a) Given the assignment `x, y, z = 1, 2, 3`, what do `x, y,` and `z` contain?

(b) What do `x, y,` and `z` contain after executing: `z, x, y = y, z, x?`

**7:**

*Identifiers.* Which of the following are valid Python identifiers? If not, why not? Of the invalid ones, which are keywords?

```
int32           40XL        char  $aving$  printf      print
_print          a do        this  self     __name__    0x40L
boolean         python      big-daddy      2hot2touch  type
thisIsn'tAVar   thisIsAVar  R_U_Ready                  yes
if              no          counter-1      access      -
```

The remaining problems deal with the `fgrepwc.py` application.

**8:**

In the `fgrepwc.py` program above, you will notice the use of the `string.find()` module. What does this function do, and what are its return values for success and failure?

**9:**

We briefly discussed module names above with regards to the `__name__` variable. What are the contents of this variable if we ran `fgrepwc.py` directly? What would the contents be if we imported `fgrepwc` as a module?

**10:**

The "-i" option is indicated in the `usage()` function of the `fgrepwc` module but is not implemented anywhere in the entire application. This option is to perform the search in a case-insensitive manner. Implement this functionality for `fgrepwc.py`. You may use the `getopt` module.

**11:**

`fgrepwc.py` currently outputs the number of matching lines which contain the search string. Update the script so that it outputs the total number of times the

string appears in the text file. In other words, if a match occurs more than once on a line, count all of those additional appearances.

# Chapter 4. Python Objects

We will now begin our journey to the core part of the language. First we will introduce what Python objects are, then discuss the most commonly-used built-in types. An introduction to the standard type operators and built-in functions comes next, followed by an insightful discussion of the different ways to categorize the standard types to gain a better understanding of how they work, and finally, we will conclude by describing some types that Python does *not* have (mostly as a benefit for those of you with experience with another high-level language).

## Python Objects

Python uses the object model abstraction for data storage. Any construct which contains any type of value is an object. Although Python is classified as an "object-oriented programming language," OOP is not required to create perfectly working Python applications. You can certainly write a useful Python script without the use of classes and instances. However, Python's object syntax and architecture certainly encourage or "provoke" this type of behavior. Let us now take a closer look at what a Python "object" is.

All Python objects have the following three characteristics: an *identity,* a *type,* and a *value.*

| | |
|---|---|
| IDENTITY | Unique identifier that differentiates an object from all others. Any object's identifier can be obtained using the `id()` built-in function. This value is as close as you will get to a "memory address" in Python (probably much to the relief of some of you). Even better is that you rarely, if ever, access this value, much less care what it is at all. |
| TYPE | An object's type indicates what kind of values an object can hold, what operations can be applied to such objects, and what behavioral rules these objects are subject to. You can use the `type()` built-in function to reveal the type of a Python object. Since types are also objects in Python (did we mention that Python was object-oriented?), `type()` actually returns an object to you rather than a simple literal. |
| VALUE | Data item that is represented by an object. |

All three are assigned on object creation and are read-only with one exception, the value. If an object supports updates, its value can be changed; otherwise, it is also read-only. Whether an object's value can be changed is known as an object's *mutability,* which we will investigate later on in Section 4.7. These characteristics hang around as long as the object does and are reclaimed when an object is deallocated.

Python supports a set of basic (built-in) data types, as well as some auxiliary types that may come into play if your application requires them. Most applications generally use the standard types and create and instantiate classes for all specialized data storage.

### Object Attributes

Certain Python objects have attributes, data values or executable code such as methods, associated with them. Attributes are accessed in the dotted attribute notation, which includes the name of the associated object, and were introduced in the Core Note near Section 3.14. The most familiar attributes are functions and methods, but some Python types have data attributes associated with them. Objects with data attributes include (but are not limited to): classes, class instances, modules, complex numbers, and files.

## Standard Types

Numbers (four separate sub-types)

Regular or "Plain" Integer

Long Integer

Floating Point Real Number

Complex Number

String

List

Tuple

Dictionary

We will also refer to standard types as "primitive data types" in this text because these types represent the primitive data types that Python provides. We will go over each one in detail in Chapters 5, 6 and 7.

**NOTE**

In Java, although primitive data types are supported, they usually come in class "wrappers" for which instances are created when a data type is needed. In Python,

standard types are not classes, so creating integers and strings does not involve instantiation. That also means that you cannot subclass a standard type either, although there is nothing wrong with wrapping a type around a Python class and modifying a class to what you desire. Python also provides some classes which emulate types and can be subclassed. See Section 13.18.

## Other Built-in Types

Type

None

File

Function

Module

Class

Class Instance

Method

These are some of the other types you will interact with as you develop as a Python programmer. We will also cover these in Chapters 9, 11, 12, and 13 with the exception of the `Type` and `None` types, which we will discuss here.

### Types and the `type()` Built-in Function

It may seem unusual perhaps, to regard types themselves as objects since we are attempting to just describe all of Python's types to you in this chapter. However, if you keep in mind that an object's set of inherent behaviors and characteristics (such as supported operators and built-in methods) must be defined somewhere, an object's type is a logical place for this information. The amount of information necessary to describe a type cannot fit into a single string; therefore types cannot simply be strings, nor should this information be stored with the data, so we are back to types as objects.

We will formally introduce the `type()` built-in function. The syntax is as follows:

```
type(object)
```

The `type()` built-in function takes object and returns its type. The return object is a `type` object.

```
>>> type(4)                 #int type
<type 'int'>
>>>
>>> type('Hello World!')    #string type
<type 'string'>
>>>
>>> type(type(4))           #type type
<type 'type'>
```

In the examples above, we take an integer and a string and obtain their types using the `type()` built-in function; in order to also verify that types themselves are types, we call `type()` on the output of a `type()` call.

Note the interesting output from the `type()` function. It does not look like a typical Python data type, i.e., a number or string, but is something enclosed by greater-than and less-than signs. This syntax is generally a clue that what you are looking at is an object. Objects may implement a printable string representation; however, this is not always the case. In these scenarios where there is no easy way to "display" an object, Python "pretty-prints" a string representation of the object. The format is usually of the form: *`<object_something_or_another>`*. Any object displayed in this manner generally gives the object type, an object ID or location, or other pertinent information.

### None

Python has a special type known as the Null object. It has only one value, `None`. The type of `None` is also `None`. It does not have any operators or built-in functions. If you are familiar with C, the closest analogy to the `None` *type* is `void`, while the `None` value is similar to the C value of `NULL`. (Other similar objects and values include Perl's `undef` and Java's `Void` type and `null` value.) `None` has no attributes and *always* evaluates to having a Boolean *false* value.

## Internal Types

Code

Frame

Traceback

Slice

Ellipsis

Xrange

We will briefly introduce these internal types here. The general application programmer would typically not interact with these objects directly, but we include them here for completeness. Please refer to the source code or Python internal and online documentation for more information.

In case you were wondering about exceptions, they are now implemented as classes not types. In older versions of Python, exceptions were implemented as strings.

## Code Objects

Code objects are executable pieces of Python source that are byte-compiled, usually as return values from calling the `compile()` built-in function. Such objects are appropriate for execution by either **`exec`** or by the `eval()` built-in function. All this will be discussed in greater detail in Chapter 14.

Code objects themselves do not contain any information regarding their execution environment, but they are at the heart of every user-defined function, all of which *do* contain some execution context. (The actual byte-compiled code as a code object is one attribute belonging to a function). Along with the code object, a function's attributes also consist of the administrative support which a function requires, including its name, documentation string, default arguments, and global namespace.

## Frames

These are objects representing execution stack frames in Python. Frame objects contain all the information the Python interpreter needs to know during a runtime execution environment. Some of its attributes include a link to the previous stack frame, the code object (see above) that is being executed, dictionaries for the local and global namespaces, and the current instruction. Each function call results in a new frame object, and for each frame object, a C stack frame is created as well. One place where you can access a frame object is in a traceback object (see below).

## Tracebacks

When you make an error in Python, an exception is raised. If exceptions are not caught or "handled," the interpreter exits with some diagnostic information similar to the output shown below:

```
Traceback (innermost last):
  File "<stdin>", line N?, in ???
ErrorName:  error reason
```

The traceback object is just a data item that holds the stack trace information for an exception and is created when an exception occurs. If a handler is provided for an exception, this handler is given access to the traceback object.

## Slice Objects

Slice objects are created when using the Python extended slice syntax. This extended syntax allows for different types of indexing. These various types of indexing include *stride indexing,* multi-dimensional indexing, and indexing using the Ellipsis type. The syntax for multi-dimensional indexing is `sequence[start1 : end1, start2 : end2]`, or using the ellipsis, `sequence[…, start1 : end1]`. Slice objects can also be generated by the `slice()` built-in function. Extended slice syntax is currently supported only in external third party modules such as the NumPy module and JPython.

Stride indexing for sequence types allows for a third slice element that allows for "step"-like access with a syntax of `sequence[starting_index : ending_index : stride]`. We will demonstrate an example of stride indexing using JPython here:

```
% jpython
JPython 1.1 on java1.1.8 (JIT: sunwjit)
Copyright (C) 1997-1999 Corporation for National Research
Initiatives
>>> foostr = 'abcde'
>>> foostr[::-1]
'edcba'
>>> foostr[::-2]
'eca'
>>> foolist = [123, 'xba', 342.23, 'abc']
>>> foolist[::-1]
['abc', 342.23, 'xba', 123]
```

## Ellipsis

Ellipsis objects are used in extended slice notations as demonstrated above. These objects are used to represent the actual ellipses in the slice syntax (…). Like the Null object, ellipsis objects also have a single name, `Ellipsis`, and has a Boolean *true* value at all times.

## Xranges

XRange objects are created by the built-in function `xrange()`, a sibling of the `range()` built-in function and used when memory is limited and for when `range()` generates an unusually large data set. You can find out more about `range()` and `xrange()` in Chapter 8.

For an interesting side adventure into Python types, we invite the reader to take a look at the `types` module in the standard Python library.

**NOTE**

All standard type objects can be tested for truth value and compared to objects of the same type. Objects have inherent ***true*** or ***false*** values. Objects take a false value when they are empty, any numeric representation of zero, or the Null object `None`.

The following are defined as having false values in Python:

`None`

Any numeric zero:

`0` ([plain] integer)

`0.0` (float)

`0L` (long integer)

`0.0+0.0j` *(complex)*

`""` (empty string)

`[]` (empty list)

`()` (empty tuple)

`{}` (empty dictionary)

Any value for an object other than the those above is considered to have a true value, i.e., non-empty, non-zero, etc. User-created class instances have a false value when their nonzero ( `__nonzero__()` ) or length ( `__len__()` ) special methods, if defined, return a zero value.

## Standard Type Operators

### Value Comparison

Comparison operators are used to determine equality of two data values between members of the same type. These comparison operators are supported for all built-in types. Comparisons yield true or false values, based on the validity of the comparison expression. Python chooses to interpret these values as the plain integers 0 and 1 for false and true, respectively, meaning that each comparison will result in one of those two possible values. A list of Python's value comparison operators is given in Table 4.1.

**Table 4.1. Standard Type Value Comparison Operators**

| operator | function |
|---|---|

| | |
|---|---|
| `expr1 < expr2` | *expr1* is less than *expr2* |
| `expr1 > expr2` | *expr1* is greater than *expr2* |
| `expr1 <= expr2` | *expr1* is less than or equal to *expr2* |
| `expr1 >= expr2` | *expr1* is greater than or equal to *expr2* |
| `expr1 == expr2` | *expr1* is equal to *expr2* |
| `expr1 != expr2` | *expr1* is not equal to *expr2* (C-style) |
| `expr1 <> expr2` | *expr1* is not equal to *expr2* (ABC/Pascal-style)[a] |

[a] This "not equals" sign will slowly be phased out. Use != instead.

Note that comparisons performed are those that are appropriate for each data type. In other words, numeric types will be compared according to numeric value in sign and magnitude, strings will compare lexicographically, etc.

```
>>> 2 == 2
1
>>> 2.46 <= 8.33
1
>>> 5+4j >= 2-3j
1
>>> 'abc' == 'xyz'
0
>>> 'abc' > 'xyz'
0
>>> 'abc' < 'xyz'
1
>>> [3, 'abc'] == ['abc', 3]
0
>>> [3, 'abc'] == [3, 'abc']
1
```

Also, unlike many other languages, multiple comparisons can be made on the same line, evaluated in left-to-right order:

```
>>> 3 < 4 < 7        # same as ( 3 < 4 ) and ( 4 < 7 )
1
>>> 4 > 3 == 3       # same as ( 4 > 3 ) and ( 3 == 3 )
1
>>> 4 < 3 < 5 != 2 < 7
0
```

We would like to note here that comparisons are strictly between object values, meaning that the comparisons are between the data values and not the actual data objects themselves. For the latter, we will defer to the object identity comparison operators described next.

## Object Identity Comparison

In addition to value comparisons, Python also supports the notion of directly *comparing objects* themselves. Objects can be assigned to other variables (by reference). Because each variable points to the same (shared) data object, any change effected through one variable will change the object and hence be reflected through all references to the same object.

In order to understand this, you will have to think of variables as linking to objects now and be less concerned with the values themselves. Let us take a look at three examples.

### Example 1: foo1 and foo2 reference the same object

```
foo1 = foo2 = 4
```

When you look at this statement from the value point-of-view, it appears that you are performing a multiple assignment and assigning the numeric value of 4 to both the `foo1` and `foo2` variables. This is true to a certain degree, but upon lifting the covers, you will find that a numeric object with the contents or value of 4 has been created. Then that object's reference is assigned to both `foo1` and `foo2`, resulting in both `foo1` and `foo2` aliased to the same object. Figure 4-1 shows an object with two references.

**Figure 4-1. `foo1` and `foo2` Reference the Same Object**


**Example 2: foo1 and foo2 reference the same object**

```
foo1 = 4
foo2 = foo1
```

This example is very much like the first: A numeric object with value 4 is created, then assigned to one variable. When `foo2 = foo1` occurs, `foo2` is directed to the same object as `foo1` since Python deals with objects by passing references. `foo2` then becomes a new and additional reference for the original value. So both `foo1` and `foo2` now point to the same object. The same figure above applies here as well.

**Example 3: foo1 and foo2 reference different objects**

```
foo1 = 4
foo2 = 1 + 3
```

This example is different. First, a numeric object is created, then assigned to `foo1.` Then a second numeric object is created, and this time assigned to `foo2.` Although both objects are storing the exact same value, there are indeed two distinct objects in the system, with `foo1` pointing to the first, and `foo2` being a reference to the second. Figure 4-2 below shows now we have two distinct objects even though both objects have the same value.

**Figure 4-2. `foo1` and `foo2` Reference Different Objects**

Why did we choose to use boxes in our diagrams above? Well, a good way to visualize this concept is to imagine a box (with contents inside) as an object. When a variable is assigned an object, that creates a "label" to stick on the box, indicating a reference has been made. Each time a new reference to the same object is made, another sticker is put on the box. When references are abandoned, then a label is removed. A box can be "recycled" only when all the labels have been peeled off the box. How does the system keep track of how many labels are on a box?

Each object has associated with it a counter that tracks the total number of references that exist to that object. This number simply indicates how many variables are "pointing to" any particular object. This is the *reference count* that we introduced in the last chapter in Sections 3.5.5–3.5.7. Python provides the **is** and **is not** operators to test if a pair of variables do indeed refer to the same object. Performing a check such as

```
a is b
```

is an equivalent expression to

```
id(a) == id(b)
```

The object identity comparison operators all share the same precedence level and are presented in Table 4.2.

**Table 4.2. Standard Type Object Identity Comparison Operators**

| operator | function |
|---|---|
| *obj1* **is** obj2 | *obj1* is the same object as *obj2* |
| *obj1* **is not** *obj2* | *obj1* is not the same object as *obj2* |

In the example below, we create a variable, then another that points to the same object.

```
>>> a = [ 5, 'hat', -9.3]
>>> b = a
>>> a is b
1
>>> a is not b
0
>>>
>>> b = 2.5e-5
```

```
>>> b
2.5e-005
>>> a
[5, 'hat', -9.3]
>>> a is b
0
>>> a is not b
1
```

Both the **is** and **not** identifiers are Python keywords.

## Boolean

Expressions may be linked together or negated using the boolean logical operators **and**, **or**, and **not**, all of which are Python keywords. These Boolean operations are in highest-to-lowest order of precedence in Table 4.3. The **not** operator has the highest precedence and is immediately one level below all the comparison operators. The **and** and **or** operators follow, respectively.

**Table 4.3. Standard Type Boolean Operators**

| operator | function |
|---|---|
| **not** *expr* | logical NOT of *expr* (negation) |
| *expr1* **and** *expr2* | logical AND of *expr1* and *expr2* (conjunction) |
| *expr1* **or** *expr2* | logical OR of *expr1* and *expr2* (disjunction) |

```
>>> x, y = 3.1415926536, -1024
>>> x < 5.0
1
>>> not (x < 5.0)
0
>>> (x < 5.0) or (y > 2.718281828)
1
>>> (x < 5.0) and (y > 2.718281828)
0
>>> not (x is y)
1
```

Earlier, we introduced the notion that Python supports multiple comparisons within one expression. These expressions have an implicit **and** operator joining them together.

```
>>> 3 < 4 < 7      # same as "( 3 < 4 ) and ( 4 < 7 )"
1
```

## Standard Type Built-in Functions

Along with generic operators which we have just seen, Python also provides some built-in functions that can be applied to all the basic object types: `cmp(), repr(), str(), type(),` and the single reverse or back quotes ( `''` ) operator, which is functionally-equivalent to `repr()`.

**Table 4.4. Standard Type Built-in Functions**

| function | operation |
|---|---|
| `cmp(obj1, obj2)` | compares `obj1` and `obj2`, returns integer `i` where:<br>`i < 0` if `obj1 < obj2`<br>`i > 0` if `obj1 > obj2`<br>`i == 0` if `obj1 == obj2` |
| `repr(obj)/' obj'` | returns evaluatable string representation of `obj` |
| `str(obj)` | returns printable string representation of `obj` |
| `type(obj)` | determines type of `obj` and return type object |

### cmp()

The `cmp()` built-in function CoMPares two objects, say, `obj1` and `obj2`, and returns a negative number (integer) if `obj1` is less than `obj2`, a positive number if `obj1` is greater than `obj2`, and zero if `obj1` is equal to `obj2`. Notice the similarity in return values as C's `strcmp()`. The comparison used is the one that applies for that type of object, whether it be a standard type or a user-created class; if the latter, `cmp()` will call the class's special `__cmp__()` method. More on these special methods in Chapter 13, on Python classes. Here are some samples of using the `cmp()` built-in function with numbers and strings.

```
>>> a, b = -4, 12
>>> cmp(a,b)
```

```
-1
>>> cmp(b,a)
1
>>> b = -4
>>> cmp(a,b)
0
>>>
>>> a, b = 'abc', 'xyz'
>>> cmp(a,b)
-23
>>> cmp(b,a)
23
>>> b = 'abc'
>>> cmp(a,b)
0
```

We will look at using `cmp()` with other objects later.

## `str()` and `repr()` (and `''` Operator)

The `str()` STRing and `repr()` REPResentation built-in functions or the single back or reverse quote operator ( `` ` ` `` ) come in really handy if the need arises to either recreate an object through evaluation or obtain a human-readable view of the contents of objects, data values, object types, etc. To use these operations, a Python object is provided as an argument and some type of string representation of that object is returned.

In some examples below, we take some random Python types and convert them to their string representations.

```
>>> str(4.53-2j)
'(4.53-2j)'
>>>
>>> str(1)
'1'
>>>>>> str(2e10)
'20000000000.0'
>>>
>>> str([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>>
>>> repr([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>>
>>> `[0, 5, 9, 9]`
'[0, 5, 9, 9]'
```

Although all three are similar in nature and functionality, only `repr()` and `` ` ` `` do exactly the same thing, and using them will deliver the "official" string representation of an

object that can be evaluated as a valid Python expression (using the `eval()` built-in function). In contrast, `str()` has the job of delivering a "printable" string representation of an object which may not necessarily be acceptable by `eval()`, but will look nice in a **`print`** statement.

The executive summary is that `repr()` is Python-friendly while `str()` produces human-friendly output. However, with that said, because both types of string representations coincide so often, on many occasions all three return the exact same string.

**NOTE**

Occasionally in Python, you will find both an operator and a function that do exactly the same thing. One reason why both an operator and a function exist is that there are times where a function may be more useful than the operator, for example, when you are passing around executable objects like functions and where different functions may be called depending on the data item. Another example is the double-star ( `**` ) and `pow()` built-in function which performs "`x` to the `y` power" exponentiation for `x ** y` or `pow(x,y)`.

## A Second Look at `type()`

Python does not support method or function overloading, so you are responsible for any "introspection" of the objects that your functions are called with. (Also see the Python FAQ 4.75.) Fortunately, we have the `type()` built-in function to help us with just that, introduced earlier in Section 4.3.1.

What's in a name? Quite a lot, if it is the name of a type. It is often advantageous and/or necessary to base pending computation on the type of object that is received. Fortunately, Python provides a built-in function just for that very purpose. `type()` returns the type for any Python object, not just the standard types. Using the interactive interpreter, let's take a look at some examples of what `type()` returns when we give it various objects.

```
>>> type('')
<type 'string'>
>>>
>>> s = 'xyz'
>>>
>>> type(s)
<type 'string'>
>>>
>>> type(100)
<type 'int'>
>>>
>>> type(-2)
```

```
<type 'int'>
>>>
>>> type(0)
<type 'int'>
>>>
>>> type(0+0j)
<type 'complex'>
>>>
>>> type(0L)
<type 'long int'>
>>>
>>> type(0.0)
<type 'float'>
>>>
>>> type([])
<type 'list'>
>>>
>>> type(())
<type 'tuple'>
>>>
>>> type({})
<type 'dictionary'>
>>>
>>> type(type)
<type 'builtin_function_or_method'>
>>>
>>> type(Abc)
<type 'class'>
>>>
>>> type(Abc_obj)
<type 'instance'>
```

You will find most of these types familiar, as we discussed them at the beginning of the chapter, however, you can now see how Python recognizes types with `type()`. Since we cannot usually "look" at a type object to reveal its value from outside the interactive interpreter, the best use of the type object is to compare it with other type objects to make this determination.

```
>>> type(1.2e-10+4.5e20j) == type(0+0j):
1
>>> type('this is a string') == type(''):
1
>>> type(34L) == type(0L)
1
>>> type(2.4) == type(3)
0
```

Although `type()` returns a type object rather than an integer, say, we can still use it to our advantage because we can make a direct comparison using the **`if`** statement. We

present below a script in Example 4.1 that shows how we can use `type()` in a run-time environment.

### Example 4.1. Checking the Type (`typechk.py`)

The function `displayNumType()` takes a numeric argument and uses the `type()` built-in to indicate its type (or "not a number," if that is the case).

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   def displayNumType(number):
004 4
005 5       print number, "is",
006 6       if type(number) == type(0):
007 7           print 'an integer'
008 8       elif type(number) == type(0L):
009 9           print 'a long'
010 10      elif type(number) == type(0.0):
011 11          print 'a float'
012 12      elif type(number) == type(0+0j):
013 13          print 'a complex number'
014 14      else: <$nopage>
015 15          print 'not a number at all!!'
016 16
017 17   displayNumType(-69)
018 18   displayNumType(99999999999999999999L)
019 19   displayNumType(98.6)
020 20   displayNumType(-5.2+1.9j)
021 21   displayNumType('xxx')
022  <$nopage>
```

Running `typechk.py`, we get the following output:

```
-69 is an integer
99999999999999999999 is a long
98.6 is a float
(-5.2+1.9j) is a complex number
xxx is not a number at all!!
```

An alternative to comparing an object's type with a known object's type (as we did above and in the example below) is to utilize the `types` module which we briefly mentioned earlier in the chapter.

```
>>> import types
>>> aFloat = 1.69
>>> if type(aFloat) == types.FloatType:
…       print aFloat, 'is a float type'
… else:
…       print aFloat, 'is not a float type'
```

```
…
1.69 is a float type
```

A summary of operators and built-in functions common to all basic Python types is given in Table 4.5. The progressing shaded groups indicate hierarchical precedence from highest-to-lowest order. Elements grouped with similar shading all have equal priority.

**Table 4.5. Standard Type Operators and Built-In Functions**

| operator/function | description | *result*[a] |
|---|---|---|
| string representation | | |
| `` `` | string representation | string |
| built-in functions | | |
| `cmp(obj1, obj2)` | compares two objects | integer |
| `repr(obj)` | string representation | string |
| `str(obj)` | string representation | string |
| `type(obj)` | determines object type | type object |
| value comparisons | | |
| `<` | less than | Boolean |
| `>` | greater than | Boolean |
| `<=` | less than or equal to | Boolean |

| | | |
|---|---|---|
| >= | greater than or equal to | Boolean |
| == | equal to | Boolean |
| != | not equal to | Boolean |
| <> | not equal to | Boolean |
| object comparisons | | |
| **is** | the same as | Boolean |
| **is not** | not the same as | Boolean |
| Boolean operators | | |
| **not** | logical negation | Boolean |
| **and** | logical conjuction | Boolean |
| **or** | logical disjunction | Boolean |

[a] Those results labelled as "Boolean" indicate a Boolean comparison; since Python does not have a Boolean type per se, the result returned is a plain integer with value 0 (for false) or 1 (for true).

## Categorizing the Standard Types

If we were to be maximally verbose in describing the standard types, we would probably call them something like Python's "basic built-in data object primitive types."

"Basic," indicating that these are the standard or core types that Python provides

"Built-in," due to the fact that types these come default with Python. (We use this term very loosely so as to not confuse them with Python built-in variables and functions.)

"Data," because they are used for general data storage

"Object," because objects are the default abstraction for data and functionality

"Primitive," because these types provide the lowest-level granularity of data storage

"Types," because that's what they are: data types!

However, this description does not really give you an idea of how each type works or what functionality applies to them. Indeed, some of them share certain characteristics, such as how they function, and others share commonality with regards to how their data values are accessed. We should also be interested in whether the data that some of these types hold can be updated and what kind of storage they provide.

There are three different models we have come up with to help categorize the standard types, with each model showing us the interrelationships between the types. These models help us obtain a better understanding of how the types are related, as well as how they work.

## Storage Model

The first way we can categorize the types is by how many objects can be stored in an object of this type. Python's types, as well as types from most other languages, can hold either single or multiple values. A type which holds a single object we will call *literal* or *scalar* storage, and those which can hold multiple objects we will refer to as *container* storage. (Container objects are also referred to as *composite* or *compound* objects in the documentation, but some of these refer to objects other than types, such as class instances.) Container types bring up the additional issue of whether different types of objects can be stored. All of Python's container types can hold objects of different types. Table 4.6 categorizes Python's types by storage model.

**Table 4.6. Types categorized by the Storage Model**

| Storage model category | Python types that fit category |
|---|---|
| literal/scalar | numbers (all numeric types), strings |
| container | lists, tuples, dictionaries |

Although strings may seem like a container type since they "contain" characters (and usually more than one character), they are not considered as such because Python does not have a character type (see Section 4.8). Thus, because strings are self-contained literals.

## Update Model

Another way of categorizing the standard types is by asking the question, "Once created, can objects be changed or their values updated?" When we introduced Python types early on, we indicated that certain types allow their values to be updated and others do not. Mutable objects are those whose values can be changed, and immutable objects are those whose values cannot be changed. Table 4.7 illustrates which types support updates and which do not.

**Table 4.7. Types Categorized by the Update Model**

| Update model category | Python types that fit category |
|---|---|
| mutable | lists, dictionaries |
| immutable | numbers, strings, tuples |

Now after looking at the table, a thought which must immediately come to mind is, "Wait a minute! What do you mean that numbers and strings are immutable? I've done things like the following:"

```
x = 'Python numbers and strings'
x = 'are immutable?!? What gives?'
i = 0
i = i + 1
```

"They sure as heck don't look immutable to me!" That is true to some degree, but looks can be deceiving. What is really happening behind the scenes is that the original objects are actually being replaced in the above examples. Yes, that is right. Read that again.

Rather than pointing to the original objects, new objects with the new values were allocated and (re)assigned to the original variable names, and the old objects were garbage-collected. One can confirm this by using the `id()` built-in function to compare object identities before and after such assignments.

If we added calls to `id()` in our example above, we may be able to see that the objects are being changed, as below:

```
x = 'Python numbers and strings'
print id(x)
x = 'are immutable?!? What gives?'
print id(x)
i = 0
```

```
print id(i)
i = i + 1
print id(i)
```

Upon executing our little piece of code, we get the following output. Your mileage will vary since object IDs will differ from system to system and are memory location dependent:

```
16191392
16191232
7749552
7749600
```

On the flip side, lists can be modified without replacing the original object, as illustrated in the code below.

```
>>> aList = [ 'ammonia', 83, 85, 'lady' ]
>>> aList
['ammonia', 83, 85, 'lady']
>>>
>>> aList[2]
85
>>>
>>> id(aList)
135443480
>>>
>>> aList[2] = aList[2] + 1
>>> aList[3] = 'stereo'
>>> aList
['ammonia', 83, 86, 'stereo']
>>>
>>> id(aList)
135443480
>>>
>>> aList.append('gaudy')
>>> aList.append(aList[2] + 1)
>>> aList
['ammonia', 83, 86, 'stereo', 'gaudy', 87]
>>>
>>> id(aList)
135443480
```

Notice how for each change, the ID for the list remained the same.

## Access Model

Although the previous two models of categorizing the types are useful when being introduced to Python, they are not the primary models for differentiating the types. For that purpose, we use the access model. By this, we mean, how do we access the values of our stored data? There are three categories under the access model: *direct, sequence,* and *mapping.* The different access models and which types fall into each respective category are given in Table 4.8.

**Table 4.8. Types Categorized by the Access Model**

| access model category | types that fit category |
|---|---|
| direct | numbers |
| sequence | strings, lists, tuples |
| mapping | dictionaries |

Direct types indicate single element, non-container types. All numeric types fit into this category.

Sequence types are those whose elements are sequentially-accessible via index values starting at 0. Accessed items can be either single elements or in groups, better known as slices. Types which fall into this category include strings, lists, and tuples. As we mentioned before, Python does not support a character type, so, although strings are literals, they are a sequence type because of the ability to access substrings sequentially.

Mapping types are similar to the indexing properties of sequences, except instead of indexing on a sequential numeric offset, elements (values) are unordered and accessed with a key, thus making mapping types a set of hashed key-value pairs.

We will use this primary model in the next chapter by presenting each access model type and what all types in that category have in common (such as operators and built-in functions), then discussing each Python standard type that fits into those categories. Any operators, built-in functions, and methods unique to a specific type will be highlighted in their respective sections.

So why this side trip to view the same data types from differing perspectives? Well, first of all, why categorize at all? Because of the high-level data structures that Python provides, we need to differentiate the "primitive" types from those which provide more functionality. Another reason is to be clear on what the expected behavior of a type should be. For example, if we minimize the number of times we ask ourselves, "What are the differences between lists and tuples again?" or "What types are immutable and which are not?" then we have done our job. And finally, certain categories have general

characteristics which apply to all types which belong to a certain category. A good craftsman (and craftswoman) should know what is available in his or her toolboxes.

The second part of our inquiry asks, why all these different models or perspectives? It seems that there is no one way of classifying all of the data types. They all have crossed relationships with each other, and we feel it best to expose the different sets of relationships shared by all the types. We also want to show how each type is unique in its own right. No two types map the same across all categories. And finally, we believe that understanding all these relationships will ultimately play an important implicit role during development. The more you know about each type, the more you are apt to use the correct ones in the parts of your application where they are the most appropriate, and where you can maximize performance.

We summarize by presenting a cross-reference chart (see Table 4.9) which shows all the standard types, the three different models we use for categorization, and where each type fits into these models.

**Table 4.9. Categorizing the Standard Types**

| Data Type | Storage Model | Update Model | Access Model |
|---|---|---|---|
| numbers | literal/scalar | immutable | direct |
| strings | literal/scalar | immutable | sequence |
| lists | container | mutable | sequence |
| tuples | container | immutable | sequence |
| dictionaries | container | mutable | mapping |

## Unsupported Types

Before we explore each standard type, we conclude this chapter by giving a list of types that are not supported by Python.

### Boolean

Unlike Pascal or Java, Python does not feature the Boolean type. Use integers instead.

### char or byte

Python does not have a char or byte type to hold either single character or 8-bit integers. Use strings of length one for characters and integers for 8-bit numbers.

### pointer

Since Python manages memory for you, there is no need to access pointer addresses. The closest to an address that you can get in Python is by looking at an object's identity using the `id()` built-in function. Since you have no control over this value, it's a moot point.

### int vs. short vs. long

Python's plain integers are the universal "standard" integer type, obviating the need for three different integer types, i.e., C's `int, short,` and `long.` For the record, Python's integers are implemented as C `long`s. For values larger in magnitude than regular integers (usually your system architecture size, i.e., 32-bit), use Python's `long` integer.

### float vs. double

C has both a single precision `float` type and double-precision `double` type. Python's `float` type is actually a C `double.` Python does not support a single-precision floating point type because its benefits are outweighed by the overhead required to support two types of floating point types.

## Exercises

**1:**

*Python Objects.* What three values are associated with *all* Python objects?

**2:**

*Types.* Which Python types are immutable?

**3:**

*Types.* Which Python types are sequences?

**4:**

*`type()` Built-in Function.* What does the `type()` built-in function do? What kind of object does `type()` return—an integer or an object?

**5:**

*`str()` and `repr()` Built-in Functions.* What are the differences between the `str()` and `repr()` built-in functions and the backquote ( `` ) operator?

**6:**

*Object Equality.* What do you think is the difference between the expressions `type(a) == type(b)` and `type(a) is type(b)`?

**7:**

`dir()` *Built-in Function.* In Exercises 2-12 and 2-13, we experimented with a built-in function called `dir()` which takes an object and reveals its attributes. Do the same thing for the `types` module. Write down the list of the types that you are familiar with, including all you know about each of these types; then create a separate list of those you are not familiar with. As you learn Python, deplete the "unknown" list so that all of them can be moved to the "familiar with" list.

# Chapter 5. Numbers

In this chapter, we will focus on Python's numeric types. We will cover each type in detail, then present the various operators and built-in functions which can be used with numbers. We conclude this chapter by introducing some of the standard library modules which deal with numbers.

## Introduction to Numbers

Numbers provide literal or scalar storage and direct access. Numbers are also an immutable type, meaning that changing or updating its value results in a newly allocated object. This activity is, of course, transparent to both the programmer and the user, so it should not change the way the application is developed.

Python has four types of numbers: "plain" integers, long integers, floating point real numbers, and complex numbers.

### How to Create and Assign Numbers (Number Objects)

Creating numbers is as simple as assigning a value to a variable:

```
anInt = 1
1aLong = -9999999999999999L
aFloat = 3.1415926535897932384626433832795
aComplex = 1.23 + 4.56J
```

### How to Update Numbers

You can "update" an existing number by (re)assigning a variable to another number. The new value can be related to its previous value or to a completely different number altogether.

```
anInt = anInt + 1
aFloat = 2.718281828
```

### How to Remove Numbers

Under normal circumstances, you do not really "remove" a number; you just stop using it! If you really want to delete a reference to a number object, just use the **del** statement (introduced in Section 3.5.6). You can no longer use the variable name, once removed, unless you assign it to a new object; otherwise, you will cause a NameError exception to occur.

```
                                        del anInt
del aLong, aFloat, aComplex
```

Okay, now that you have a good idea of how to create and update numbers, let us take a look at Python's four numeric types.

## Integers

Python has two types of integers. Plain integers are the generic vanilla (32-bit) integers recognized on most systems today. Python also has a long integer size; however, these far exceed the size provided by C longs. We will take a look at both types of Python integers, followed by a description of operators and built-in functions applicable only to Python integer types.

### (Plain) Integers

Python's "plain" integers are the universal numeric type. Most machines (32-bit) running Python will provide a range of $-2^{31}$ to $2^{31}-1$, that is -2,147,483,648 to 2,147,483,647. Here are some examples of Python integers:

```
0101    84    -237    0x80    017    -680    -0X92
```

Python integers are implemented as (signed) longs in C. Integers are normally represented in base 10 decimal format, but they can also be specified in base eight or base sixteen representation. Octal values have a "0" prefix, and hexadecimal values have either "0x" or "0X" prefixes.

### Long Integers

The first thing we need to say about Python long integers is to *not* get them confused with long integers in C or other compiled languages—these values are typically restricted to 32- or 64-bit sizes, whereas Python long integers are limited only by the amount of (virtual) memory in your machine. In other words, they can be very L-O-N-G longs.

Long integers are a superset of integers and are useful when the range of plain integers exceeds those of your application, meaning less than $-2^{31}$ or greater than $2^{31}-1$. Use of

long integers is denoted by an upper- or lowercase (L) or (l), appended to the integer's numeric value. Values can be expressed in decimal, octal, or hexadecimal. The following are examples of long integers:

```
16384L       -0x4E8L 017L  -21474836481 052144364L

2997924581   0xDECADEDEADBEEFBADFEEDDEAL-5432101234L
```

**NOTE**

Although Python supports a case-insensitive "L" to denote long integers, we recommend that you use only the uppercase "L" to avoid confusion with the number "one" ( 1 ). Python will display only long integers with a capital "L."

```
>>> aLong = 9999999991

>>> aLong

999999999L
```

## Floating Point Real Numbers

Floats in Python are implemented as C `double`s, double precision floating point real numbers, values which can be represented in straightfoward decimal or scientific notations. These 8-byte (64-bit) values conform to the IEEE 754 definition (52M/11E/1S) where 52 bits are allocated to the mantissa, 11 bits to the exponent (this gives you about $\pm 10^{308.25}$ in range), and the final bit to the sign. That all sounds fine and dandy; however, the actual amount of precision you will receive (along with the range and overflow handling) depends completely on the architecture of the machine as well as the implementation of the compiler which built your Python interpreter.

Floating point values are denoted by a decimal point ( `.` ) in the appropriate place and an optional "e" suffix representing scientific notation. We can use either lowercase ( `e` ) or uppercase ( `E` ). Positive (+) or negative ( `-` ) signs between the "e" and the exponent indicate the sign of the exponent. Absence of such a sign indicates a positive exponent. Here are some floating point values:

```
0.0     -777.     1.6       -5.555567119   96e3 * 1.0
4.3e25  9.384e-23 -2.172818 float(12)      1.000000001
3.1416  4.2E-10   -90.      6.022e23       -1.609E-19
```

## Complex Numbers

A long time ago, mathematicians were stumped by the following equation:

$$x^2 = -1$$

The reason for this is because any real number (positive or negative) multiplied by itself results in a positive number. How can you multiply any number with itself to get a negative number? No such real number exists. So in the eighteenth century, mathematicians invented something called an *imaginary number i* (or *j*— depending what math book you are reading) such that:

$$j = \sqrt{-1}$$

Basically a new branch of mathematics was created around this special number (or concept), and now imaginary numbers are used in numerical and mathematical applications. Combining a real number with an imaginary number forms a single entity known as a *complex number.* A complex number is any ordered pair of floating point real numbers (`x, y`) denoted by `x + y j` where `x` is the real part and `y` is the imaginary part of a complex number.

Here are some facts about Python's support of complex numbers:

Imaginary numbers by themselves are not supported in Python

Complex numbers are made up of real and imaginary parts

Syntax for a complex number: *`real+imag`* **`j`**

Both real and imaginary components are floating point values

Imaginary part is suffixed with letter "J" lowercase ( `j`) or upper (`J`)

The following are examples of complex numbers:

```
64.375+1j    4.23-8.5j    0.23-8.55j  1.23e-045+6.7e+089j
6.23+1.5j    -1.23-875J  0+1j9.80665-8. 31441J  -.0224+0j
```

## Complex Number Built-in Attributes

Complex numbers are one example of objects with data attributes (Section 4.1.1). The data attributes are the real and imaginary components of the complex number object they belong to. Complex numbers also have a method attribute which can be invoked, returning the complex conjugate of the object.

```
>>> aComplex = -8.333-1.47j
>>> aComplex
(-8.333-1.47j)
>>> aComplex.real
```

```
-8.333
>>> aComplex.imag
-1.47
>>> aComplex.conjugate()
(-8.333+1.47j)
```

Table 5.1 describes the attributes which complex numbers have:

**Table 5.1. Complex Number Attributes**

| attribute | desciption |
|---|---|
| *num*.real | real component of complex number *num* |
| *num*.imag | imaginary component of complex number *num* |
| *num*.conjugate() | returns complex conjugate of *num* |

## Operators

Numeric types support a wide variety of operators, ranging from the standard type of operators to operators created specifically for numbers, and even some which apply to integer types only.

### Mixed-Mode Operations

It may be hard to remember, but when you added a pair of numbers in the past, what was important was that you got your numbers correct. Addition using the plus ( + ) sign was always the same. In programming languages, this may not be as straightforward because there are different types of numbers.

When you add a pair of integers, the + represents integer addition, and when you add a pair of floating point numbers, the + represents double-precision floating point addition, and so on. Our little description extends even to non-numeric types in Python. For example, the + operator for strings represents concatenation, not addition, but it uses the same operator! The point is that for each data type that supports the + operator, there are different pieces of functionality to "make it all work," embodying the concept of *overloading*.

Now, we cannot add a number and a string, but Python does support mixed mode operations strictly between numeric types. When adding an integer and a float, a choice has to be made as to whether integer or floating point addition is used. There is no hybrid

operation. Python solves this problem using something called numeric coercion. This is the process whereby one of the operands is converted to the same type as the other before the operation. Python perform's *numeric coercion* by following some rules:

To begin with, if both numbers are the same type, no conversion is necessary. When both types are different, a search takes place to see whether one number can be converted to the other's type. If so, the operation occurs and both numbers are returned, one having been converted. There are rules that must be followed since certain conversions are impossible, such as turning a float into an integer, or converting a complex number to any non-complex number type.

Coercions which are possible, however, include turning an integer into a float (just add " `.0` ") or converting any non-complex type to a complex number (just add a zero imaginary component, i.e., " `0j` "). The rules of coercion follow from these two examples: integers move towards float, and all move toward complex. The Python Reference Guide describes the `coerce()` operation in the following manner:

If either argument is a complex number, the other is converted to complex;

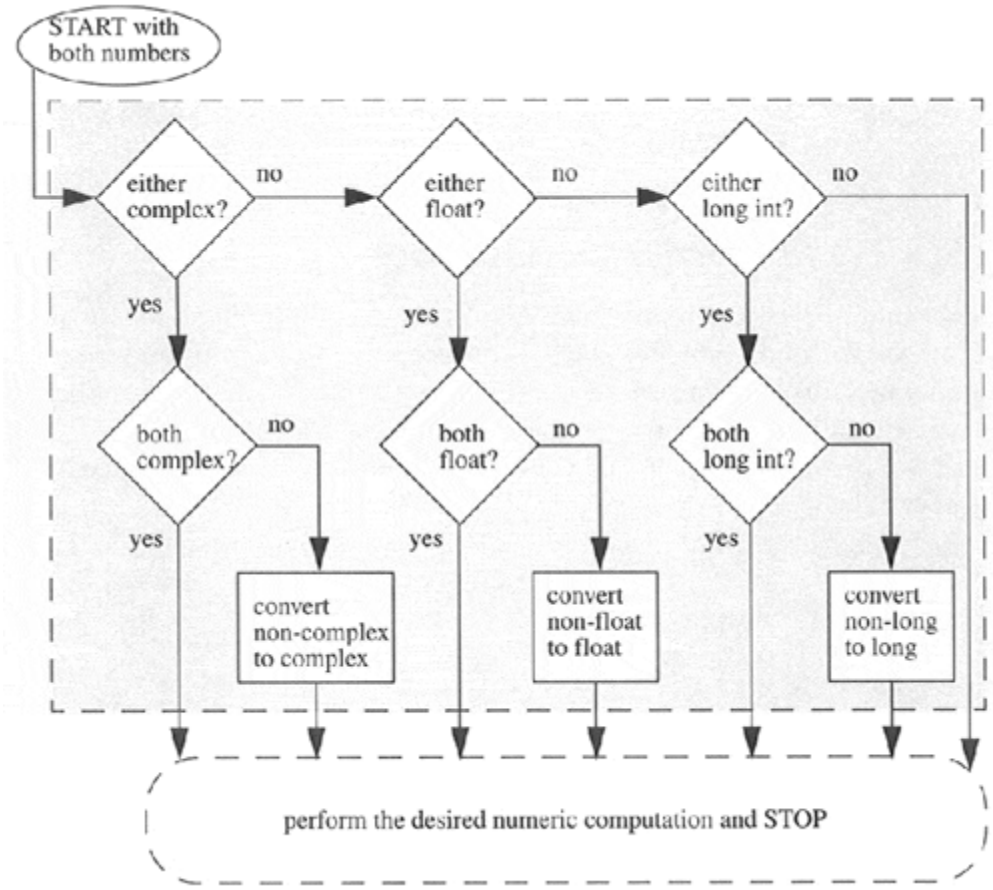
Otherwise, if either argument is a floating point number, the other is converted to floating point;

Otherwise, if either argument is a long integer, the other is converted to long integer;

Otherwise, both must be plain integers and no conversion is necessary (in the upcoming diagram, this describes the rightmost arrow).

The following flowchart illustrates these coercion rules:

**Figure 5.1. Numeric coercion.**




Automatic numeric coercion makes life easier for the programmer since he or she does not have to worry about adding coercion code to his or her application. If explicit coercion is desired, Python does provide the `coerce()` built-in function (described later in Section 5.6.2).

If there is any bad news about coercion and mixed-mode operations, it is that no coercion is performed *during* an operation. For example, if you multiply two integers together forming a result that is too large for an integer, no conversion to a long takes place, and your operation will fail:

```
>>> 999999 * 999999
Traceback (innermost last):
  File "<stdin>", line 1, in ?
OverflowError: integer multiplication
```

A workaround to such a situation is to try to detect if such problems may occur, and if so, perform a manual conversion of both integers to longs by using the `long()` built-in function before the operation.

Below is an example showing you Python's automatic coercion. The `2` is converted to a long before the operation.

```
>>> 999999L ** 2.
99999B000001L
```

## Standard Type Operators

The standard type operators discussed in the previous chapter all work as advertised for numeric types. Mixed-mode operations, described above, are those which involve two numbers of different types. The values are internally converted to the same type before the operation is applied.

Here are some examples of the standard type operators in action with numbers:

```
>>> 5.2 == 5.2
1
>>> -719 >= 833
0
>>> 5+4e >= 2-3e
1
>>> 2 < 5 < 9         # same as ( 2 < 5 ) and ( 5 < 9 )
1
>>> 77 > 66 == 66     # same as ( 77 > 66 ) and ( 66 == 66 )
1
>>> 0. < -90.4 < 55.3e2 != 3 < 181
0
>>> (-1 < 1) or (1 < -1)
1
```

## Numeric Type (Arithmetic) Operators

Python supports unary operators for no change and negation, `+` and `-`, respectively; and binary arithmetic operators `+`, `-`, `*`, `/`, `%`, and `**`, for addition, subtraction, multiplication, division, modulo, and exponentiation, respectively.

Rules and exceptions: Any zero right-hand argument for division and modulo will result in a `ZeroDivisionError` exception. Integer modulo is straightforward integer division remainder, while for float, take the difference of the dividend and the product of the divisor and the quotient of the quantity dividend divided by the divisor rounded down to the closest integer, i.e., `x - (math.floor(x/y) * y,` or

$$x - \left\lfloor \frac{x}{y} \right\rfloor \times y$$

For complex number modulo, take only the real component of the division result, i.e., `x - (math.floor((x/y).real) * y)`.

The exponentiation operator has a peculiar precedence rule in its relationship with the unary operators: It binds more tightly than unary operators to its left, but less tightly than unary operators to its right. Due to this characteristic, you will find the `**` operator twice in the numeric operator charts in this text. Here are some examples:

```
>>> 3 ** 2
9
>>> -3 ** 2      # ** binds together than - to its left
-9
>>> (-3) ** 2    # group to cause - to bind first
9>>> 4.0 ** -1.0 # ** binds looser than - to its right
0.25
```

In the second case, it performs 3 to the power of 2 (3-squared) before it applies the unary negation. We need to use the parentheses around the "-3" to prevent this from happening. In the final example, we see that the unary operator binds more tightly because the operation is 1 over quantity 4 to the first power $\frac{1}{4^1}$ or ¼. Note that 1/4 as an integer operation results in an integer 0, so integers are not allowed to be raised to a negative power (it is a floating point operation anyway), as we will show here:

```
>>> 4 ** -1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: integer to the negative power
```

A summary of all arithmetic operators, in shaded hierarchical order from highest-to-lowest priority is found Table 5.2. All the operators listed here rank higher in priority than the bitwise operators for integers, found in Section 5.5.4.

**Table 5.2. Numeric Type Arithmetic Operators**

| arithmetic operator | function |
|---|---|
| *expr1* ** *expr2* | *expr1* raised to the power of *expr2* [a] |
| + *expr* | (unary) *expr* sign unchanged |

| | |
|---|---|
| - *expr* | (unary) negation of *expr* |
| *expr1* ** *expr2* | *exp1* raised to the power of *expr2* [a] |
| *expr1* * *expr2* | *expr1* times *expr2* |
| *expr1* / *expr2* | *expr1* divided by *expr2* |
| *expr1* % *expr2* | *expr1* modulo *expr2* |
| *expr1* + *expr2* | *expr1* plus *expr2* |
| *expr1* - *expr2* | *expr1* minus *expr2* |

[a] blinds tighter than unary operators to its left and looser than unary operators to its right

Be aware that integer division truncates. To obtain the correct fractional result, use floating point numbers instead:

```
>>> 3 / 4
0
>>> 3.0 / 4.0
0.75
```

Here are a few more examples of Python's numeric operators.

```
>>> -442 - 77
-519
>>>
>>> 4 ** 3
64
>>>
>>> 4.2 ** 3.2
98.7183139527
>>> 8 / 3
2
>>> 8.0 / 3.0
2.66666666667
>>> 8 % 3
```

```
2
>>> (60. - 32.) * ( 5. / 9. )
15.5555555556
>>> 14 * 0x04
56
>>> 0170 / 4
30
>>> 0x80 + 0777
639
>>> 45L * 22L
990L
>>> 16399L + 0xA94E8L
709879L
>>> -2147483648L - 52147483648L
-54294967296L
>>> 64.375+1j + 4.23-8.5j
(68.605-7.5j)
>>> 0+1j ** 2     # same as 0+(lj**2)
(-1+0j)
>>> 1+1j ** 2     # same as 1+(lj**2)
0j
>>> (1+1j) ** 2
2j
```

Note how the exponentiation operator is still higher in priority than the binding addition operator that delimits the real and imaginary components of a complex number. Regarding the last two examples above, we grouped the components of the complex number together to obtain the desired result.

## *Bit Operators (Integer-only)

Python integers may be manipulated bitwise and the standard bit operations are supported: inversion, bitwise AND, OR, and exclusive OR (a.k.a. XOR), and left and right shifting. Here are some facts regarding the bit operators:

Negative numbers are treated as their 2's complement value.

Left and right shifts of N bits are equivalent to multiplication and division by (2 ** N) without overflow checking.

For long integers, the bit operators use a "modified" form of 2's complement, acting as if the sign bit were extended infinitely to the left.

The bit inversion operator ( `~` ) has the same precedence as the arithmetic unary operators, the highest of all bit operators. The bit shift operators ( `<<` and `>>` ) come next, having a precedence one level below that of the standard plus and minus operators, and finally we have the bitwise AND, XOR, and OR operators (`&, ^, |` ), respectively. All of the bitwise operators are presented in the order of descending priority in Table 5.3.

| Table 5.3. Integer Type Bitwise Operators | |
|---|---|
| bitwise operator | function |
| *~ num* | (unary) invert the bits of num, yielding -( *num* + 1) |
| *num1 << num2* | *expr1* left shifted by *expr2 bits* |
| *num1 >> num2* | *expr1* right shifted by *expr2 bits* |
| *num1 & num2* | *expr1* bitwise AND with *expr2* |
| *num1 ^ num2* | *expr1* bitwise XOR (exclusive OR) with *expr2* |
| *num1 \| num2* | *expr1* bitwise OR with *expr2* |

We will now present some examples using the bit operators using 30 (011110), 45 (101101), and 60 (111100):

```
>>> 30 & 45
12
>>> 30 | 45
63
>>> 45 & 60
44
>>> 45 | 60
61
>>> ~30
-31
>>> ~45
-46
>>> 45 << 1
90
>>> 60 >> 2
15
>>> 30 ^ 45
51
```

## Built-in Functions

## Standard Type Functions

In the last chapter, we introduced the `cmp(), str(),` and `type()` built-in functions that apply for all standard types. For numbers, these functions will compare two numbers, convert numbers into strings, and tell you a number's type, respectively. Here are some examples of using these functions:

```
>>> cmp(-6, 2)]
-1
>>> cmp(-4.333333, -2.718281828)
-1
>>> cmp(0xFF, 255)
0
>>> str(0xFF)
'255'
>>> str(55.3e2)
'5530.0'
>>> type(0xFF)
<type 'int'>
>>> type(98765432109876543210L)
<type 'long int'>
>>> type(2-1j)
<type 'complex'>
```

## Numeric Type Functions

Python currently supports different sets of built-in functions for numeric types. Some convert from one numeric type to another while others are more operational, performing some type of calculation on their numeric arguments.

### Conversion

The `int(), long(), float(),` and `complex()` built-in functions are used to convert from any numeric type to another. Starting in Python 1.5, these functions will also take strings and return the numerical value represented by the string.

The following are some examples using the numeric type conversion built-ins:

```
>>> int(4.25555)
4
>>> long(42)
42L
>>> float(4)
4.0
>>> complex(4)
(4+0j)
>>>
>>> complex(2.4, -8)
```

```
(2.4-8j)
>>>
>>> complex(2.3e-10, 45.3e4)
(2.3e-10+453000j)
```

Table 5.4 nutshells these numeric type conversion built-in functions.

**Table 5.4. Numeric Type Conversion Built-in Functions**

| function | operation |
|---|---|
| `int( obj, base=10 )` | converts string or number *obj* to (plain) integer; provides same behavior as `string.atoi()`; optional base argument introduced in 1.6 |
| `long( obj, base=10 )` | converts string or number *obj* to long integer; provides same behavior as `string.atol()`; optional *base* argument introduced in 1.6 |
| `float( obj )` | converts string or number *obj* to floating point; provides same behavior as `string.atof()` |
| `complex( str )` **or** `complex( real, imag=0.0)` | converts string *str* to complex, or takes *real* (and perhaps *imag* inary) numbers and returns a complex number with those components |

### Operational

Python has five operational built-in functions for numeric types: `abs()`, `coerce()`, `divmod()`, `pow()`, and `round()`. We will take a look at each and present some usage examples.

`abs()` returns the absolute value of the given argument. If the argument is a complex number, then `math.sqrt( num.real`$^2$ `+ num.imag`$^2$`)` is returned. Here are some examples of using the `abs()` built-in function:

```
>>> abs(-1)
1
>>> abs(10.)
10.0
>>> abs(1.2-2.1j)
```

```
2.41867732449
>>> abs(0.23 - 0.78)
0.55
```

The `coerce()` function, although it technically is a numeric type conversion function, does not convert to a specific type and acts more like an operator, hence our placement of it in our operational built-ins section. In Section 5.5.1, we discussed numeric coercion and how Python performs that operation. The `coerce()` function is a way for the programmer to explicitly coerce a pair of numbers rather than letting the interpreter do it. This feature is particularly useful when defining operations for newly-created numeric class types. `coerce()` just returns a tuple containing the converted pair of numbers. Here are some examples:

```
>>> coerce(1, 2)
(1, 2)
>>>
>>> coerce(1.3, 134L)
(1.3, 134.0)
>>>
>>> coerce(1, 134L)
(1L, 134L)
>>>
>>> coerce(1j, 134L)
(1j, (134+0j))
>>>
>>> coerce(1.23-41j, 134L)
((1.23-41j), (134+0j))
```

The `divmod()` built-in function combines division and modulus operations into a single function call that returns the pair (quotient, remainder) as a tuple. The values returned are the same as those given for the standalone division and modulus operators for integer types. For floats, the quotient returned is `math.floor( num1/num2 )` and for complex numbers, the quotient is `math.floor(( num1/num2 ).real)`.

```
>>> divmod(10,3)
(3, 1)
>>> divmod(3,10)
(0, 3)
>>> divmod(10,2.5)
(4.0, 0.0)
>>> divmod(2.5,10)
(0.0, 2.5)
>>> divmod(2+1j, 0.5-1j)
(0j, (2+1j))
```

Both `pow()` and the double star ( `**` ) operator perform exponentiation; however, there are differences other than the fact that one is an operator and the other is a built-in function.

The `**` operator did not appear until Python 1.5, and the `pow()` built-in takes an optional third parameter, a modulus argument. If provided, `pow()` will perform the exponentiation first, then return the result modulo the third argument. This feature is used for cryptographic applications and has better performance than `pow(x,y) % z` since the latter performs the calculations in Python rather than in C like `pow(x, y, z)`.

```
>>> pow(2,5)
32
>>>
>>> pow(5,2)
25
>>> pow(3.141592,2)
9.86960029446
>>>
>>> pow(1+1j, 3)
(-2+2j)
```

The `round()` built-in function has a syntax of round `(flt,ndig=0)`. It normally rounds a floating point number to the nearest integral number and returns that result (still) as a float. When the optional third `ndig` option is given, `round()` will round the argument to the specific number of decimal places.

```
>>> round(3)
3.0
>>> round(3.45)
3.0
>>> round(3.4999999)
3.0
>>> round(3.4999999, 1)
3.5
>>> import math
>>> for eachNum in range(10):
…     print round(math.pi, eachNum)
…
3.0
3.1
3.14
3.142
3.1416
3.14159
3.141593
3.1415927
3.14159265
3.141592654
3.1415926536
>>> round(-3.5)
```

```
-4.0
>>> round(-3.4)
-3.0
>>> round(-3.49)
-3.0
>>> round(-3.49, 1)
-3.5
```

Note that the rounding performed by `round()` moves away from zero on the number line, i.e., `round(.5)` goes to 1 and `round(-.5)` goes to -1. Also, with functions like `int()`, `round()`, and `math.floor()`, all may seem like they are doing the same thing; it is possible to get them all confused. Here is how you can differentiate among these:

`int()` chops off the decimal point and everything after (a.k.a. truncation).

`floor()` rounds you to the next smaller integer, i.e., the next integer moving in a negative direction (towards the left on the number line).

`round()` (rounded zero digits) rounds you to the nearest integer period.

Here is the output for four different values, positive and negative, and the results of running these three functions on eight different numbers. (We reconverted the result from `int()` back to a float so that you can visualize the results more clearly when compared to the output of the other two functions.)

```
>>> import math
>>> for eachNum in (.2, .7, 1.2, 1.7, -.2, -.7, -1.2, -1.7):
…     print "int(%.1f)\t%+.1f" % (eachNum, float(int(eachNum)))
…     print "floor(%.1f)\t%+.1f" % (eachNum,
…     math.floor(eachNum))
…     print "round(%.1f)\t%+.1f" % (eachNum, round(eachNum))
…     print '-' * 20
…
int(0.2)         +0.0
floor(0.2)       +0.0
round(0.2)       +0.0
--------------------
int(0.7)         +0.0
floor(0.7)       +0.0
round(0.7)       +1.0
--------------------
int(1.2)         +1.0
floor(1.2)       +1.0
round(1.2)       +1.0
--------------------
int(1.7)         +1.0
floor(1.7)       +1.0
round(1.7)       +2.0
--------------------
int(-0.2)        +0.0
```

```
floor(-0.2)       -1.0
round(-0.2)       +0.0
--------------------
int(-0.7)         +0.0
floor(-0.7)       -1.0
round(-0.7)       -1.0
--------------------
int(-1.2)         -1.0
floor(-1.2)       -2.0
round(-1.2)       -1.0
--------------------
int(-1.7)         -1.0
floor(-1.7)       -2.0
round(-1.7)       -2.0
```

Table5.5 summarizes the operational functions for numeric types:

**Table 5.5. Numeric Type Operational Built-in Functions**[a]

| function | operation |
|---|---|
| `abs( num )` | returns the absolute value of *num* |
| `coerce( num1, num2 )` | converts *num1* and *num2* to the same numeric type and returns the converted pair as a tuple |
| `divmod( num1, num2 )` | division-modulo combination returns *(num1 / num2, num1 % num2)* as a tuple. For floats and complex, the quotient is rounded down (complex uses only real component of quotient) |
| `pow( num1, num2, mod =1)` | raises *num1* to *num2* power, quantity modulo mod if provided |
| `round( flt, ndig =0)` | (floats only) takes a float *flt* and rounds it to *ndig* digits, defaulting to zero if not provided |

[a] except for `round()`, which applies only to floats

## Integer-only Functions

In addition to the built-in functions for all numeric types, Python supports a few that are specific only to integers (plain and long). These functions fall into two categories, base presentation with `hex()` and `oct()`, and ASCII conversion featuring `chr()` and `ord()`.

### Base Representation

As we have seen before, Python integers automatically support octal and hexadecimal representations in addition to the decimal standard. Also, Python has two built-in functions which return string representations of an integer's octal or hexadecimal equivalent. These are the `oct()` and `hex()` built-in functions, respectively. They both take an integer (in any representation) object and return a string with the corresponding value. The following are some examples of their usage:

```
>>> hex(255)
'0xff'
>>> hex(230948231)
'0x1606627L'
>>> hex(65535*2)
'0x1fffe'
>>>
>>> oct(255)
'0377'
>>> oct(230948231)
'0130063047L'
>>> oct(65535*2)
'0377776'
```

### ASCII Conversion

Python also provides functions to go back and forth between ASCII (American Standard Code for Information Interchange) characters and their ordinal integer values. Each character is mapped to a unique number in a table numbered from 0 to 255. This number does not change for all computers using the ASCII table, providing consistency and expected program behavior across different systems. `chr()` takes a single-byte integer value and returns a one-character string with the equivalent ASCII character. `ord()` does the opposite, taking a single ASCII character in the form of a string of length one and returns the corresponding ASCII value as an integer:

```
>>> ord('a')
97
>>> ord('A')
65
>>> ord('0')
48

>>> chr(97)
'a'
>>> chr(65L)
```

```
'A'
>>> chr(48)
'0'
```

Table 5.6 shows all built-in functions for integer types.

**Table 5.6. Integer Type Built-in Functions**

| function | operation |
|---|---|
| `hex( num )` | converts *num* to hexadecimal and return as string |
| `oct( num )` | converts *num* to octal and return as string |
| `chr( num )` | takes ASCII value *num* and returns ASCII character as string; 0 <= *num* <= 255 only |
| `ord( chr )` | takes ASCII *chr* and returns corresponding ordinal ASCII value; *chr* must be a string of length 1 |

## Related Modules

There are a number of modules in the Python standard library that add-on to the functionality of the operators and built-in functions for numeric types. Table 5.7 lists the key modules for use with numeric types. Refer to the literature or online documentation for more information on these modules.

**Table 5.7. Numeric Type Related Modules**

| module | contents |
|---|---|
| `array` | implements array types… a restricted sequence type |
| `math/cmath` | supplies standard C library mathematical functions; most functions available in `math` are implemented for complex numbers in the `cmath` module |

| | |
|---|---|
| `operator` | contains numeric operators available as function calls, i.e., `operator.sub(m, n)` is equivalent to the difference `(m - n)` for numbers `m` and `n` |
| `random` | is default RNG module for Python… obsoletes `rand` and `whrandom` |

For advanced numerical and scientific mathematics applications, there is also a well known external module called `NumPy` which may be of interest to you.

**NOTE**

The `random` module is the general-purpose place to go if you are looking for random numbers. The random number generator (RNG), based on the Wichmann-Hill algorithm, comes seeded with the current timestamp and is ready to go as soon as it has loaded. Here are four of the most commonly used functions in the `random` module:

| | |
|---|---|
| `randint()` | takes two integer values and returns a random integer between those values inclusive |
| `uniform()` | does almost the same thing as `randint()`, but returns a float and is inclusive only of the smaller number (exclusive of the larger number) |
| `random()` | works just like `uniform()` except that the smaller number is fixed at 0.0, and the larger number is fixed at 1.0 |
| `choice()` | given a sequence (see *Chapter 6*), randomly selects and returns a sequence item |

We have now come to the conclusion of our tour of all of Python's numeric types. A summary of operators and built-in functions for numeric types is given in Table 5.8.

**Table 5.8. Operators and Built-in Functions for All Numeric Types**

| Operator/built-in | Description | int | long | float | complex | *Result* [a] |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| `abs()` | absolute value | • | • | • | • | numbera[a] |
| `chr()` | character | • | • | | | string |
| `coerce()` | numeric coercion | • | • | • | • | tuple |
| `complex()` | complex conversion | • | • | • | • | complex |
| `divmod()` | division/modulo | • | • | • | • | tuple |
| `float()` | float conversion | • | • | • | • | float |
| `hex()` | hexadecimal string | • | • | | | string |
| `int()` | int conversion | • | • | • | • | int |
| `long()` | long conversion | • | • | • | • | long |
| `oct()` | octal string | • | • | | | string |
| `ord()` | ordinal | | | (string) | | int |
| `pow()` | exponentiation | • | • | • | • | number |
| `round()` | float rounding | | | • | | float |
| `**` [b] | exponentiation | • | • | • | • | number |
| `+` [c] | no change | • | • | • | • | number |

| | | | | | | |
|---|---|---|---|---|---|---|
| - [c] | negation | • | • | • | • | number |
| ~ [c] | bit inversion | • | • | | | int/long |
| ** [b] | exponentiation | • | • | • | • | number |
| * | multiplication | • | • | • | • | number |
| / | division | • | • | • | • | number |
| % | modulo/remainder | • | • | • | • | number |
| + | addition | • | • | • | • | number |
| - | subtraction | • | • | • | • | number |
| << | bit left shift | • | • | | | int/long |
| >> | bit right shift | • | • | | | int/long |
| & | bitwise AND | • | • | | | int/long |
| ^ | bitwise XOR | • | • | | | int/long |
| \| | bitwise OR | • | • | | | int/long |

[a] a result of "number" indicates any of the four numeric types

[b] has a unique relationship with unary operators; see Section 5.5.3 and Table 5.2

[c] **unary operator**

## Exercises

The exercises in this chapter may first be implemented as applications. Once full functionality and correctness have been verified, we recommend that the reader convert his or her code to functions which can be used in future exercises. On a related note, one style suggestion is to not use `print` statements in functions. Instead, have the functions return the appropriate value and have the caller perform any output desired. This keeps the code adaptable and reusable.

**1:**

*Integers.* Name the differences between Python's plain and long integers.

**2:**

*Operators.* (a) Create a function to take two numbers (any type) and output their sum.

(b) Write another function, but output the product of two given numbers.

**3:**

*Standard Type Operators.* Take test score input from the user and output letter grades according to the following grade scale/curve:

A: 90 – 100

B: 80 – 89

C: 70 – 79

D: 60 – 69

F: < 60

**4:**

*Modulus.* Determine whether a given year is a leap year, using the following formula: a leap year is one that is divisible by four, but not by one hundred, unless it is also divisible by four hundred. For example, 1992, 1996, and 2000 are leap years, but 1967 and 1900 are not. The next leap year falling on a century is 2400.

**5:**

*Modulus.* Calculate the number of basic American coins given a value less than 1 dollar. A penny is worth 1 cent, a nickel is worth 5 cents, a dime is worth 10 cents, and a quarter is worth 25 cents. It takes 100 cents to make 1 dollar. So given an amount less than 1 dollar (if using floats, convert to integers for this exercise), calculate the number of each type of coin necessary to achieve the amount, maximizing the number of larger denomination coins. For example, given $0.76, or 76 cents, the correct output would be "3 quarters and 1 penny." Output such as "76

pennies" and "2 quarters, 2 dimes, 1 nickel, and 1 penny" are not acceptable.

**6:**

*Arithmetic.* Create a calculator application. Write code that will take two numbers and an operator in the format: N1 OP N2, where N1 and N2 are floating point or integer values, and OP is one of the following: `+, -, *, /, %, **`, representing addition, subtraction, multiplication, division, modulus/remainder, and exponentiation, respectively, and displays the result of carrying out that operation on the input operands.

**7:**

*Sales Tax.* Take a monetary amount (i.e., floating point dollar amount [or whatever currency you use]), and determine a new amount figuring all the sales taxes you must pay where you live.

**8:**

*Geometry.* Calculate the area and volume of:

(a) squares and cubes

(b) circles and spheres

**9:**

*Style.* Answer the following numeric format questions:

(a) Why does 17 + 32 give you 49, but 017 + 32 give you 47 and 017 + 032 give you 41, as indicated in the examples below?

```
>>> 17 + 32
49
>>> 017+ 32
47
>>> 017 + 032
41
```

(b) Why does 56l + 78l give you 134L and not 1342, as indicated in the example below?

```
 >>> 56l + 78l
134L
```

**10:**

*Conversion.* Create a pair of functions to convert Fahrenheit to Celsius temperature values. `C = (F - 32) * (5 / 9)` should help you get started.

**11:**

*Modulus.* (a) Using loops and numeric operators, output all even numbers from 0 to 20.

(b) Same as part (a), but output all odd numbers up to 20.

(c) From parts (a) and (b), what is an easy way to tell the difference between even and odd numbers?

(d) Using part (c), write some code to determine if one number divides another. In your solution, ask the user for both numbers and have your function answer "yes" or "no" as to whether one number divides another by returning 1 or 0, respectively.

**12:**

*Limits.* Determine the largest and smallest ints, longs, floats, and complex numbers that your system can handle.

**13:**

*Conversion.* Write a function that will take a time period measured in hours and minutes and return the total time in minutes only.

**14:**

*Bank account interest.* Create a function to take an interest percentage rate for a bank account, say, a Certificate of Deposit (CD). Calculate and return the Annual Percentage Yield (APY) if the account balance was compounded daily.

**15:**

*GCD and LCM.* Determine the greatest common divisor and least common multiple of a pair of integers.

**16:**

*Home finance.* Take an opening balance and a monthly payment. Using a loop, determine remaining balances for succeeding months, including the final payment. "Payment 0" should just be the opening balance and schedule monthly payment amount. The output should be in a schedule format similar to the following (the numbers used in this example are for illustrative purposes only):

```
Enter opening balance: 100.00
```

```
Enter monthly payment: 16.13

         Amount  Remaining
Pymt#     Paid    Balance
-----    ------  ---------
  0      $ 0.00  $100.00
  1      $16.13  $ 83.87
  2      $16.13  $ 67.74
  3      $16.13  $ 51.61
  4      $16.13  $ 35.48
  5      $16.13  $ 19.35
  6      $16.13  $  3.22
  7      $ 3.22  $  0.00
```

**17:**

**Random numbers.* Read up on the random module and do the following problem: Generate a list of a random number (1 < N <= 100) of random numbers (0 <= n <= 231 -1). Then randomly select a set of these numbers (1 <= N <= 100), sort them, and display this subset.

# Chapter 6. Sequences: Strings, Lists, and Tuples

The next family of Python types we will be exploring are those whose items are ordered and sequentially accessible via index offsets into the set. This group, known as *sequences,* includes the types: strings, lists, and tuples. We will first describe the general and common features followed by a closer examination of each type. We will first introduce all operators and built-in functions that apply to sequence types, then cover each sequence type individually. For each sequence type, we will provide the following information:

Introduction

Operators

Built-in Functions

Built-in Methods (if applicable)

Special Features (if applicable)

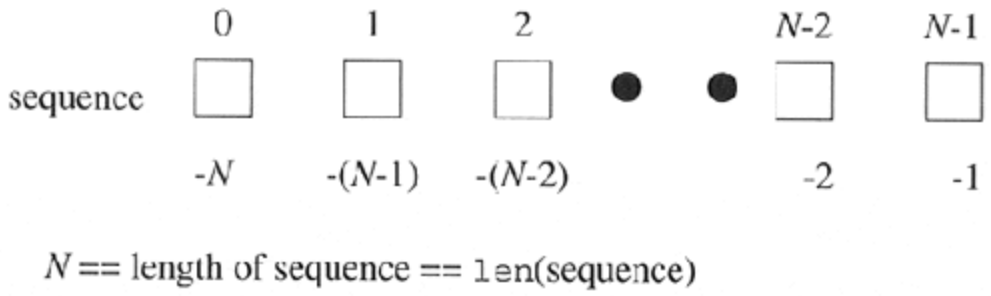
Related Modules (if applicable)

We will conclude this chapter with a reference chart that summarizes all the operators and built-in functions which apply to all sequence types. Let us begin by taking a high-level overview and examine the operators and built-in functions applicable to all sequence types.

## Sequences

Sequence types all share the same access model: ordered set with sequentially-indexed offsets to get to each element. Multiple elements may be achieved by using the slice operators which we will explore in this chapter. The numbering scheme used starts from zero (0) and ends with one less the length of the sequence—the reason for this is because we began at 0. Figure6-1 illustrates how sequence items are stored.

**Figure 6.1. How Sequence Elements Are Stored and Accessed**




## Operators

A list of all the operators applicable to all sequence types is given in Table6.1. The operators appear in hierarchical order from highest to lowest with the levels alternating between shaded and unshaded.

**Table 6.1. Sequence Type Operators**

| *Sequence Operator* | *Function* |
|---|---|
| `seq[ind]` | element located at index `ind` of `seq` |
| `seq[ind1:ind2]` | elements from index `ind1` to `ind2` of `seq` |
| `seq * expr` | `seq` repeated `expr` times |
| `seq1 + seq2` | concatenates sequences `seq1` and `seq2` |
| `obj in seq` | tests if `obj` is a member of sequence `seq` |
| `obj not in seq` | tests if `obj` is not a member of sequence `seq` |

### Membership (`in, not in`)

Membership test operators are used to determine whether an element is *in* or is a member of a sequence. For strings, this test is whether a character is in a string, and for lists and tuples, it is whether an object is an element of those sequences. The **`in`** and **`not in`** operators are Boolean in nature; they return the integer one if the membership is confirmed and zero otherwise.

The syntax for using the membership operators is as follows:

```
obj [ not ] in
                                    sequence
```

### Concatenation (`+`)

This operation allows us to take one sequence and join it with another sequence of the same type. The syntax for using the concatenation operator is as follows:

```
sequence1 + sequence2
```

The resulting expression is a new sequence which contains the combined contents of sequences *sequence1* and *sequence2.*

### Repetition (*)

The repetition operator is useful when consecutive copies of sequence elements are desired. The syntax for using the membership operators is as follows:

```
sequence * copies_int
```

The number of copies, `copies_int,` must be a plain integer. It cannot even be a long. As with the concatenation operator, the object returned is newly allocated to hold the contents of the multiply-replicated objects.

Starting in Python 1.6, `copies_int` can also be a long.

### Slices ([ ], [ : ])

Sequences are structured data types whose elements are placed sequentially in an ordered manner. This format allows for individual element access by index offset or by an index range of indices to "grab" groups of sequential elements in a sequence. This type of access is called *slicing,* and the slicing operators allow us to perform such access.

The syntax for accessing an individual element is:

```
sequence[index]
```

*sequence* is the name of the sequence and index is the offset into the sequence where the desired element is located. Index values are either positive, ranging from 0 to the length of the sequence less one, i.e., `0 <= index <= len(sequence ) -1,` or negative, ranging from -1 to the negative length of the sequence, `-len(sequence),` i.e., `-len(sequence) <= index <= -1.` The difference between the positive and negative indexes is that positive indexes start from the beginning of the sequences and negative indexes begin from the end.

Accessing a group of elements is similar. Starting and ending indexes may be given, separated by a colon (`:`). The syntax for accessing a group of elements is:

```
sequence [ [starting_index]: [ending_index]]
```

Using this syntax, we can obtain a "slice" of elements in sequence from the *starting_index* up to but not including the element at the *ending_index* index. Both *starting_index* and *ending_index* are optional, and if not provided, the slice will go from the beginning of the sequence or until the end of the sequence, respectively.

In Figures 6-2 to 6-6, we take an entire sequence (of soccer players) of length 5, and explore how to take various slices of such a sequence.

**Figure 6.2. Entire sequence: `sequence` or `sequence [ : ]`**


**Figure 6.3. Sequence slice: `sequence [0:3]` or `sequence [:3]`**

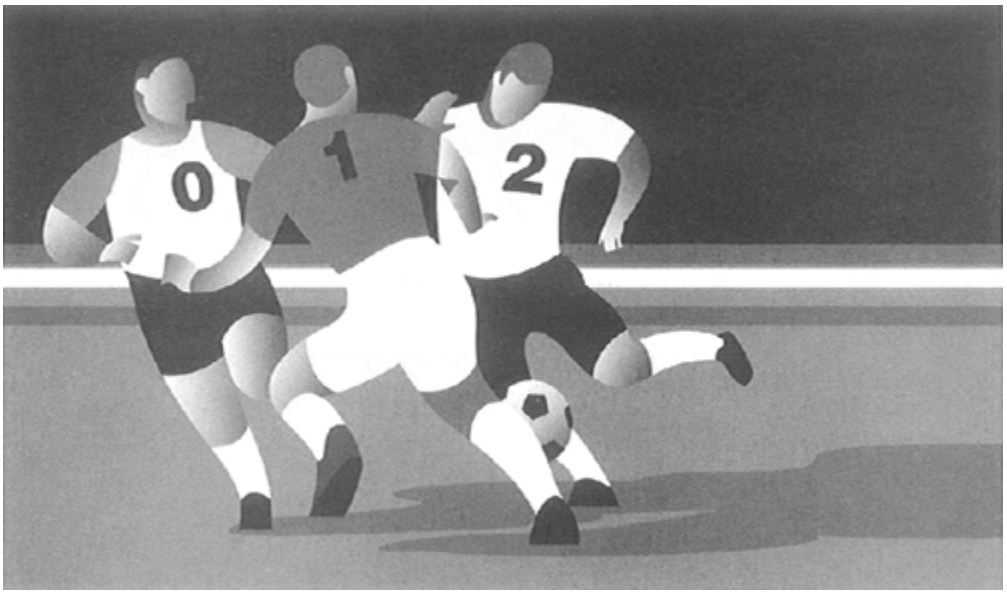


**Figure 6.4. Sequence slice: `sequence [2:5]` or `sequence [2:]`**

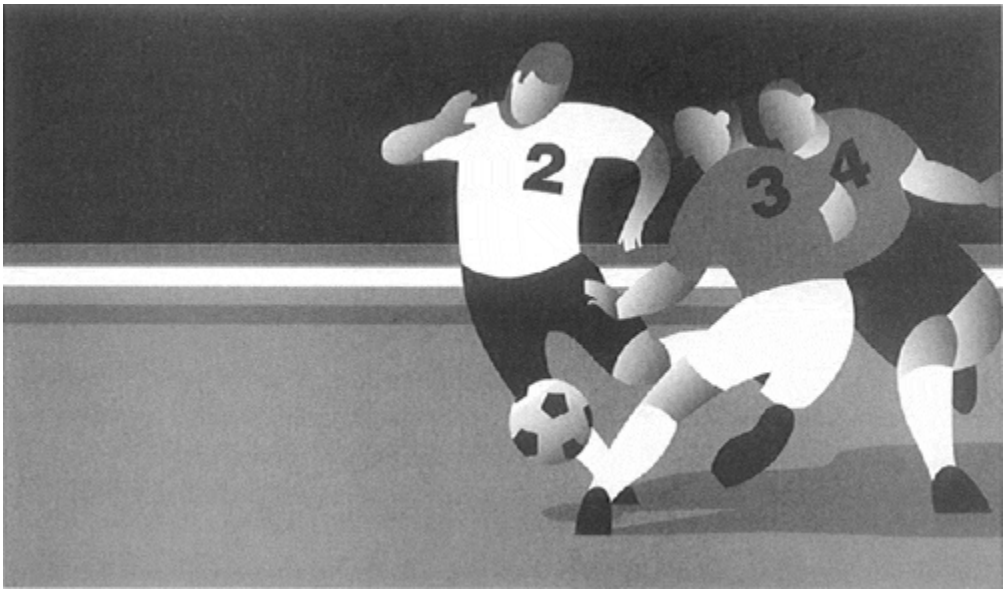



**Figure 6.5. Sequence slice: `sequence [1:3]`**


**Figure 6.6. Sequence slice: `sequence [3]`**

We will take a closer look at slicing when we cover each sequence type.

## Built-in Functions

### Conversion

The `list()`, `str()`, and `tuple()` built-in functions are used to convert from any sequence type to another. Table 6.2 lists the sequence type conversion functions.

**Table 6.2. Sequence Type Conversion Built-in Functions**

| *Function* | *Operation* |
|---|---|
| `list (seq)` | converts *seq* to list |
| `str (obj)` | converts *obj* to string |
| `tuple (seq)` | converts *seq* to tuple |

We use the term "convert" loosely. It does not actually convert the argument object into another type; recall that once Python objects are created, we cannot change their identity or their type. Rather, these functions just create a new sequence of the requested type, populate it with the members of the argument object, and pass that new sequence back as the return value. This follows a similar vein to the concatenation and repetition operations described in Section 6.1.1.

The `str()` function is most popular when converting an object into something printable and works with other types of objects, not just sequences. The `list()` and `tuple()` functions are useful to convert from one to another (lists to tuples and vice versa). However, although those functions are applicable for strings as well since strings are sequences, using `tuple()` and `list()` to turn strings into tuples or lists is not common practice.

### Operational

Python provides the following operational built-in functions for sequence types (see Table 6.3).

We are now ready to take a tour through each sequence type and will start our journey by taking a look at Python strings.

**Table 6.3. Sequence Type Operational Built-in Functions**

| *Function* | *Operation* |
|---|---|
| `len (seq)` | returns length (number of items) of *seq* |
| `max (seq)` | returns "largest" element in *seq* |
| `min (seq)` | returns "smallest" element in *seq* |

## Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. This contrasts with most other scripting languages, which use single quotes for literal strings and double quotes to allow escaping of characters. Python uses the "raw string" operator to create literal quotes, so no differentiation is necessary. Other languages such as C use single quotes for characters and double quotes for strings. Python does not have a character type; this is probably another reason why single and double quotes are the same.

Nearly every Python application uses strings in one form or another. Strings are a literal or scalar type, meaning they are treated by the interpreter as a singular value and are not containers which hold other Python objects. Strings are immutable, meaning that changing an element of a string requires creating a new string. Strings are made up of individual characters, and such elements of strings may be accessed sequentially via slicing.

### How to Create and Assign Strings

Creating strings is as simple as assigning a value to a variable:

```
>>> aString = 'Hello World!'
>>> anotherString = "Python is cool!"
>>> print aString
Hello World!
>>> print anotherString
Python is cool!
>>> aBlankString = ''
>>> print aBlankString
''
```

### How to Access Values(Characters and Substrings) in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

```
>>> aString = 'Hello World!'
>>> aString[0]
'H'
>>> aString[1:5]
'ello'
>>> aString[6:]
'World!'
```

## How to Update Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

```
>>> aString = aString[:6] + 'Python!'
>>> aString
'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether'
```

Like numbers, strings are not mutable, so you cannot change an existing string without creating a new one from scratch. That means that you cannot update individual characters or substrings in a string. However, as you can see above, there is nothing wrong with piecing together part of your old string and assigning it to a new string.

## How to Remove Characters and Strings

To repeat what we just said, strings are immutable, so you cannot remove individual characters from an existing string. What you can do, however, is to empty the string, or to put together another string which drops the pieces you were not interested in.

Let us say you want to remove one letter from "Hello World!"… the (lowercase) letter "l," for example:

```
>>> aString = 'Hello World!'
>>> aString = aString[:3] + aString[4:]
>>> aString
'Helo World!'
```

To clear or remove a string, you assign an empty string or use the **del** statement, respectively:

```
>>> aString = ''

>>> aString
''
>>> del aString
```

In most applications, strings do not need to be explicitly deleted. Rather, the code defining the string eventually terminates, and the string is automatically garbage-collected.

## Strings and Operators

### Standard Type Operators

In Chapter 4, we introduced a number of operators that apply to most objects, including the standard types. We will take a look at how some of those apply to strings. For a brief introduction, here are a few examples using strings:

```
>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> str1 < str2
1
>>> str2 != str3
1
>>> (str1 < str3) and (str2 == 'xyz')
0
```

When using the value comparison operators, strings are compared lexicographically (ASCII value order).

### Sequence Operators

#### Slices ([ ] and [ : ])

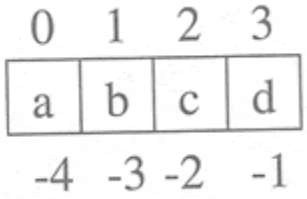
Earlier in Section 6.1.1, we examined how we can access individual or a group of elements from a sequence. We will apply that knowledge to strings in this section. In particular, we will look at:

Counting forward

Counting backward

Default/missing indexes

For the following examples, we use the single string `'abcd'`. Provided in the figure is a list of positive and negative indexes that indicate the position in which each character is located within the string itself.




Using the length operator, we can confirm that its length is 4:

```
>>> string = 'abcd'
>>> len(string)
4
```

When counting forward, indexes start at 0 to the left and end at one less than the length of the string (because we started from zero). In our example, the final index of our string is

```
final index      = len(string) - 1
                 = 4 - 1
                 = 3
```

We can access any substring within this range. The slice operator with a single argument will give us a single character, and the slice operator with a range, i.e., using a colon (`:`), will give us multiple consecutive characters. Again, for any ranges `[start:end]`, we will get all characters starting at offset `start` up to, but not including, the character at `end`. In other words, for all characters `x` in the range `[start : end]`, `start<= x < end`.

```
 >>> string[0]
'a'
>>> string[1:3]
'bc'
>>> string[2:4]
'cd'
>>> string[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Any index outside our valid index range (in our example, 0 to 3) results in an error. Above, our access of `string[2:4]` was valid because that returns characters at indexes 2 and 3, i.e., `'c'` and `'d'`, but a direct access to the character at index 4 was invalid.

When counting backward, we start at index -1 and move toward the beginning of the string, ending at negative value of the length of the string. The final index (the first character) is located at:

```
  final index  = -len(string)
               = -4
```

```
>>> string[-1]
'd'
>>> string[-3:-1]
'bc'
>>> string[-4]
'a'
```

When either a starting or an ending index is missing, they default to the beginning or end of the string, respectively.

```
>>> string[2:]
'cd'
>>> string[1:]
'bcd'
>>> string[:-1]
'abc'
>>> string[:]
'abcd'
```

Notice how the omission of both indices gives us a copy of the entire string.

### Membership (`in`, `not in`)

The membership question asks whether a character (string of length one) appears in a string. A one is returned if that character appears in the string and zero otherwise. Note that the membership operation is not used to determine if a substring is within a string. Such functionality can be accomplished by using the string methods or string module functions `find()` or `index()` (and their brethren `rfind()` and `rindex()`).

Here are a few more examples of strings and the membership operators.

```
>>> 'c' in 'abcd'
1
>>> 'n' in 'abcd'
```

```
0
>>> 'n' not in 'abcd'
1
```

In Example 6-1, we will be using the following predefined strings found in the `string` module:

```
>>> import string
>>> string.uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
```

Example 6-1 is a small script called `idcheck.py` which checks for valid Python identifiers. As we now know, Python identifiers must start with an alphabetic character. Any succeeding characters may be alphanumeric. The example also shows use of the string concatenation operator ( `+` ) introduced later in this section.

Running this script several times produces the following output:

```
% python idcheck.py
Welcome to the Identifier Checker v1.0
Testees must be at least 2 chars long.
Identifier to test? counter
okay as an identifier
%
% python idcheck.py
Welcome to the Identifier Checker v1.0
Testees must be at least 2 chars long.
Identifier to test? 3d_effects
invalid: first symbol must be alphabetic
```

Let us take apart the application line by line:

### Lines 3–6

Import the `string` module and use some of the predefined strings to put together valid alphabetic and numeric identifier strings which we will test against.

### Example 6.1. ID Check (`idcheck.py`)

Tests for identifier validity. First symbol must be alphabetic and remaining symbols must be alphanumeric. This tester program only checks identifiers which are at least two characters in length.

```
 <$nopage>
001 1  #!usr/bin/env python
002 2
003 3  import string
004 4
005 5  alphas = string.letters + '_'
006 6  nums = string.digits
007 7
008 8  print 'Welcome to the Identifier Checker v1.0'
009 9  print 'Testees must be at least 2 chars long.'
010 10 inp = raw_input('Identifier to test? ')
011 11
012 12 if len(inp) > 1:
013 13
014 14     if inp[0] not in alphas:
015 15         print '''invalid: first symbol must be
016 16              alphabetic'''
017 17     else: <$nopage>
018 18         for otherChar in inp[1:]:
019 19
020 20             if otherChar not in alphas + nums:
021 21                 print '''invalid: remaining
022 22                     symbols must be alphanumeric'''
023 23                 break <$nopage>
024 24         else: <$nopage>
025 25              print "okay as an identifier"
026  <$nopage>
```

### Lines 8–12

Print the salutation and prompt for user input. The `if` statement on line twelve filters out all statements shorter than two characters in length.

### Lines 14–16

Check to see if the first symbol is alphabetic. If it is not, display the output indicating the result and perform no further processing.

### Lines 17–18

Otherwise, loop to check the other characters, starting from the second symbol to the end of the string.

### Lines 20–23

Check to see if each remaining symbol is alphanumeric. Note how we use the concatenation operator (see below) to create the set of valid characters. As soon as we

find an invalid character, display the result and perform no further pocessing by exiting the loop with **break.**

**NOTE**

In general, repeat performances of operations or functions as arguments in a loop are unproductive as far as performance is concerned.

```
while i < l
    print 'character %d is:', string[i]
```

The loop above wastes valuable time recalculating the length of string `string.` This function call occurs for each loop iteration. If we simply save this value once, we can rewrite our loop so that it is more productive.

```
length = len(string)
while i < length:
   print 'character %d is:', string[i]
```

The same applies for a loop in the application in *Example 6-1*.

```
for otherCh
  if otherChar not in alphas + nums:
      :
```

The **for** loop beginning on line 19 contains an **if** statement that concatenates a pair of strings. These strings do not change throughout the course of the application, yet this calculation must be performed for each loop iteration. If we save the new string first, we can then reference that string rather than make the same calculations over and over again:

```
alphnums = alphas + nums
for otherChar in input[1:]:
    if otherChar not in alphnums:
       :
```

**Lines 24–25**

It may be somewhat premature to show you a **for-else** loop statement, but we are going to give it a shot anyway. (For a full treatment, see Chapter 8). The **else** statement, for a **for** loop is optional and, if provided, will execute if the loop finished in completion without being "broken" out of by **break.** In our application, if all remaining symbols check out okay, then we have a valid identifier name. The result is displayed to indicate as such, completing execution.

This application is not without its flaws however. One problem is that the identifiers tested must have length greater than 1. Our application as-is is not reflective of the true range of Python identifiers, which may be of length 1. Another problem with our application is that it does not take into consideration Python keywords, which are reserved names which cannot be used for identifiers. We leave these two tasks as exercises for the reader (see Exercise 6-2.).

### Concatenation (+)

We can use the concatenation operator to create new strings from existing ones. We have already seen the concatenation operator in action above in Example 6-1. Here are a few more examples:

```
>>> 'Spanish' + 'Inquisition'
'SpanishInquisition'
>>>
>>> 'Spanish' + ' ' + 'Inquisition'
'Spanish Inquisition'
>>>
>>> s = 'Spanish' + ' ' + 'Inquisition' + ' Made Easy'
>>> s
Spanish Inquisition Made Easy'
>>>
>>> import string
>>> string.upper(s[:3] + s[20])
'SPAM'
```

The last example illustrates using the concatenation operator to put together a pair of slices from string `s`, the "Spa" from "Spanish" and the "M" from "Made." The extracted slices are concatenated and then sent to the `string.upper()` function to convert the new string to all uppercase letters.

### Repetition (*)

The repetition operator creates new strings, concatenating multiple copies of the same string to accomplish its functionality:

```
>>> 'Ni!' * 3
'Ni!Ni!Ni!'
```

```
>>>
>>> '*'*40'
'****************************************'
>>>
>>> print '-' * 20, 'Hello World!', '-' * 20
-------------------- Hello World! --------------------
>>> who = 'knights'
>>> who * 2
'knightsknights'
>>> who
'knights'
```

As with any standard operator, the original variable is unmodified, as indicated in the final examples above.

## String-only Operators

### Format Operator (%)

One of Python's coolest features is the string format operator. This operator is unique to strings and makes up for the pack of having functions from C's `printf()` family. In fact, it even uses the same symbol, the percent sign (`%`), and supports all the `printf()` formatting codes.

The syntax for using the format operator is as follows:

```
format_string % (arguments_to_convert)
```

The *`format_string`* on the left-hand side is what you would typically find as the first argument to `printf()`, the format string with any of the embedded % codes. The set of valid codes is given in Table6.4. The *`arguments_to_convert`* parameter matches the remaining arguments you would send to `printf()`, namely the set of variables to convert and display.

**Table 6.4. Format Operator Conversion Symbols**

| *Format Symbol* | *Conversion* |
|---|---|
| `%c` | character |
| `%s` | string conversion via `str()` prior to formatting |
| `%i` | signed decimal integer |
| `%d` | signed decimal integer |
| `%u` | unsigned decimal integer |
| `%o` | octal integer |
| `%x` | hexadecimal integer (lowercase letters) |
| `%X` | hexadecimal integer (UPPERcase letters) |

| | |
|---|---|
| %e | exponential notation (with lowercase 'e') |
| %E | exponential notation (with UPPERcase 'E') |
| %f | floating point real number |
| %g | the shorter of %f and %e |
| %G | the shorter of %f and %E |

Python supports two formats for the input arguments. The first is a tuple (introduced in Section 2.8, formally in 6.15), which is basically the set of arguments to convert, just like for C's printf(). The second format which Python supports is a dictionary (Chapter 7). A dictionary is basically a set of hashed key-value pairs. The keys are requested in the *format_string*, and the corresponding values are provided when the string is formatted.

Converted strings can either be used in conjunction with the **print** statement to display out to the user or saved into a new string for future processing or displaying to a graphical user interface.

Other supported symbols and functionality are listed in Table 6.5.

**Table 6.5. Format Operator Auxiliary Directives**

| *Symbol* | *Functionality* |
|---|---|
| * | argument specifies width or precision |
| - | left justification |
| + | display the sign |
| <sp> | leave a blank space before a positive number |
| # | add the octal leading zero ( '0' ) or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used. |
| 0 | pad from left with zeros (instead of spaces) |
| % | '%%' leaves you with a single literal '%' |
| (var) | mapping variable (dictionary arguments) |
| m.n. | *m* is the minimum total width and *n* is the number of digits to display after the decimal point (if appl.) |

As with C's printf(), the asterisk symbol (*) may be used to dynamically indicate the width and precision via a value in argument tuple. Before we get to our examples, one more word of caution: long integers are more than likely too large for conversion to standard integers, so we recommend using exponential notation to get them to fit.

Here are some examples using the string format operator:

### Hexadecimal Output

```
>>> "%x" % 108
'6c'
>>>
```

```
>>> "%X" % 108
'6C'
>>>
>>> "%#X" % 108
'0X6C'
>>>
>>> "%#x" % 108
'0x6c'
```

### Floating Point and Exponential Notation Output

```
>>>
>>> '%f' % 1234.567890
'1234.567890'
>>>
>>> '%.2f' % 1234.567890
'1234.57'
>>>
>>> '%E' % 1234.567890
'1.234568E+03'
>>>
>>> '%e' % 1234.567890
'1.234568e+03'
>>>
>>> '%g' % 1234.567890
'1234.57'
>>>
>>> '%G' % 1234.567890
'1234.57'
>>>
>>> "%e" % (111111111111111111111L)
'1.111111e+21'
```

### Integer and String Output

```
>>> "%+d" % 4
'+4'
>>>
>>> "%+d" % -4
'-4'
>>>
>>> "we are at %d%%" % 100
'we are at 100%'
>>>
>>> 'Your host is: %s' % 'earth'
'Your host is: earth'
>>>
>>> 'Host: %s\tPort: %d' % ('mars', 80)
'Host: marsPort: 80'
>>>
```

```
>>> num = 123
>>> 'dec: %d/oct: %#o/hex: %#X' % (num, num, num)
'dec: 123/oct: 0173/hex: 0X7B'
>>>
>>> "MM/DD/YY = %02d/%02d/%d" % (2, 15, 67)
'MM/DD/YY = 02/15/67'
>>>
>>> w, p = 'web', 'page'
>>> 'http://xxx.yyy.zzz/%s/%s.html' % (w, p)
'http://xxx.yyy.zzz/web/page.html'
```

The previous examples all use tuple arguments for conversion. Below, we show how to use a dictionary argument for the format operator:

```
>>> 'There are %(howmany)d %(lang)s Quotation Symbols' % \
…     {'lang': 'Python', 'howmany': 3}
'There are 3 Python Quotation Symbols'
```

#### Amazing Debugging Tool

The string format operator is not only a cool, easy-to-use, and familiar feature, but a great and useful debugging tool as well. Practically all Python objects have a string presentation (either evaluatable from `repr()` or `''`, or printable from `str()`). The **`print`** statement automatically invokes the `str()` function for an object. This gets even better. When you are defining your own objects, there are hooks for you to create string representations of your object such that `repr()` and `str()` (and `''` and **`print`**) return an appropriate string as output. And if worse comes to worst and neither `repr()` or `str()` is able to display an object, the Pythonic default is to at least give you something of the format:

```
<… something that is useful …>.
```

### Raw String Operator ( r / R )

The purpose of raw strings, introduced to Python in version 1.5, is to counteract the behavior of the special escape characters that occur in strings (see the subsection below on what some of these characters are). In raw strings, all characters are taken verbatim with no translation to special or non-printed characters.

This feature makes raw strings absolutely convenient when such behavior is desired, such as when composing regular expressions (see the `re` module documentation). Regular expressions (REs) are strings which define advanced search patterns for strings and usually consist of special symbols to indicate characters, grouping and matching

information, variable names, and character classes. The syntax for REs contains enough symbols already, but when you have to insert additional symbols to make special characters act like normal characters, you end up with a virtual "alphanumersymbolic" soup! Raw strings lend a helping hand by not requiring all the normal symbols needed when composing RE patterns.

The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "**r**," which precedes the quotation marks. The "**r**" can be lowercase (**r**) or uppercase (**R**) and must be placed immediately preceding the first quote mark.

```
>>> print r'\n'
\n
>>>
>>>print '\n'
>>>
>>>print r'werbac'
werbac
>>>
>>>print r'webbac\n'
webbac\n
>>>
>>> print r'fglkjfg\][123=091'
fglkjfg\\][123=091
>>>
>>> import re
>>> aFloatRE = re.compile(R'([+-]?\d+(\.\d*)?([eE][+-
]?\d+)?))
>>> match = aFloatRE.search('abcde')
>>> print "our RE matched:", match.group(1)
''
>>> match = aFloatRE.search('-1.23e+45')
>>> print 'our RE matched:', match.group(1)
'-1.23e+45'
```

## Unicode String Operator (u/U)

The Unicode string operator, uppercase (**U**) and lowercase (**u**), introduced with Unicode string support in Python 1.6, takes standard strings or strings with Unicode characters in them and converts them to a full Unicode string object. More details on Unicode strings are available in Section 6.7.4. In addition, Unicode support is available in the new string methods (Section 6.6) and the new regular expression engine. Here are some examples:

```
u'abc'          U+0061 U+0062 U+0063
u'\u1234'       U+1234
u'abc\u1234\n'  U+0061 U+0062 U+0063 U+1234 U+0012
```

The Unicode operator can also accept raw Unicode strings if used in conjunction with the raw string operator discussed in the previous section. The Unicode operator must precede the raw string operator.

```
ur 'Hello\nWorld!'
```

# Built-in Functions

## Standard Type Functions

### cmp()

As with the value comparison operators, the `cmp()` built-in function also performs a lexicographic comparison for strings.

```
>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> cmp(str1, str2)
-11
>>> cmp(str3, str1)
23
>>> cmp(str2, 'lmn')
0
```

## Sequence Type Functions

### len()

```
>>> str1 = 'abc'
>>> len(str1)
3
>>> len('Hello World!')
12
```

The `len()` built-in function returns the number of characters in the string as expected.

### max() and min()

```
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> max(str2)
'n'
```

```
>>> min(str3)
'x'
```

Although more useful with other sequence types, the `max()` and `min()` built-in functions do operate as advertised, returning the greatest and least characters (lexicographic order), respectively.

### String Type Function [`raw_input()`]

The built-in `raw_input()` function prompts the user with a given string and accepts and returns a user-input string. Here is an example using `raw_input()`:

```
>>> user_input = raw_input("Enter your name: ")
Enter your name: John Doe
>>>
>>> user_input
'John Doe'
>>>
>>> len(user_input)
8
```

Earlier, we indicated that strings in Python do not have a terminating NUL character like C strings. We added in the extra call to `len()` to show you that what you see is what you get.

## String Built-in Methods

String methods were recently added to Python, introduced in version 1.6 (and in JPython 1.1), and tweaked for 2.0. These methods are intended to replace most of the functionality in the `string` module as well as to bring new functionality to the table. Table 6.6 shows all the current methods for strings. All string methods should fully support Unicode strings. And some are applicable only to Unicode strings.

**Table 6.6. String Type Built-in Methods**

| *Value* | *Description* |
|---|---|
| *string*`.capitalize()` | capitalizes first letter of *string* |
| *string*`.center(`*width*`)` | returns a space-padded *string* with the original *string* centered to a total of *width* columns |
| *string*`.count(`*str*`, `*beg*`= 0,`*end*`=len(`*string*`))` | counts how many times *str* occurs in *string*, or in a substring of *string* if starting |

| | |
|---|---|
| | index *beg* and ending index *end* are given |
| *string*.encode(*encoding*='UTF-8', *errors*='strict')[a] | returns encoded string version of string; on error, default is to raise a ValueError unless *errors* is given with 'ignore' or 'replace'. |
| *string*.endswith(*str*, *beg*=0, *end*=len(*string*))[b] | determines if *string* or a substring of *string* (if starting index *beg* and ending index *end* are given) ends with *str*; returns 1 if so, and 0 otherwise |
| *string*.expandtabs(*tabsize*=8) | expands tabs in *string* to multiple spaces; defaults to 8 spaces per tab if *tabsize* not provided |
| *string*.find(*str*, *beg*=0 *end*=len(*string*)) | determine if *str* occurs in *string*, or in a substring of *string* if starting index *beg* and ending index *end* are given; returns index if found and -1 otherwise |
| *string*.index(*str*, *beg*=0, *end*=len(*string*)) | same as find(), but raises an exception if *str* not found |
| *string*.isalnum()[a][b][c] | returns 1 if *string* has at least 1 character and all characters are alphanumeric and 0 otherwise |
| *string*.isalpha()[a][b][c] | returns 1 if *string* has at least 1 character and all characters are alphabetic and 0 otherwise |
| *string*.isdecimal()[b][c][d] | returns 1 if *string* contains only decimal digits and 0 otherwise |
| *string*.isdigit()[b][c] | returns 1 if *string* contains only digits and 0 otherwise |
| *string*.islower()[b][c] | returns 1 if *string* has at least 1 cased character and all cased characters are in lowercase and 0 otherwise |
| *string*.isnumeric()[b][c][d] | returns 1 if *string* contains only numeric characters and 0 otherwise |
| *string*.isspace()[b][c] | returns 1 if *string* contains only whitespace characters and 0 otherwise |

| | |
|---|---|
| *string*.istitle() [b][c] | returns 1 if *string* is properly "titlecased" (see title()) and 0 otherwise |
| *string*.isupper() [b][c] | returns 1 if *string* has at least one cased character and all cased characters are in uppercase and 0 otherwise |
| *string*.join(*seq*) | merges (concatenates) the string representations of elements in sequence *seq* into a string, with separator *string* |
| *string*.ljust(*width*) | returns a space-padded *string* with the original string left-justified to a total of *width* columns |
| *string*.lower() | converts all uppercase letters in *string* to lowercase |
| *string*.lstrip() | removes all leading whitespace in *string* |
| *string*.replace(*str1, str2, num=string*.count(str1)) | replaces all occurrences of *str1* in *string* with *str2,* or at most *num* occurrences if *num* given |
| *string*.rfind(*str, beg*=0,*end*=len(*string*)) | same as find(), but search backwards in *string* |
| *string*.rindex( *str, beg*=0, *end*=len(*string*)) | same as index(), but search backwards in *string* |
| *string*.rjust(*width*) | returns a space-padded *string* with the original string right-justified to a total of *width* columns. |
| *string*.rstrip() | removes all trailing whitespace of *string* |
| *string*.split(*str*="", *num=string*.count(*str*)) | splits *string* according to delimiter *str* (space if not provided) and returns list of substrings; split into at most *num* substrings if given |
| *string*.splitlines( *num*=string.count('\n')) [b][c] | splits *string* at all (or *num*) NEWLINEs and returns a list of each line with NEWLINEs removed |
| *string*.startswith(*str, beg*=0,*end*=len(*string*)) [b][c] | determines if *string* or a substring of *string* (if starting index *beg* and ending index *end* are given) starts |

| | |
|---|---|
| | with substring *str;* returns 1 if so, and 0 otherwise |
| *string*.strip(*[obj]*) | performs both *lstrip()* and *rstrip()* on *string* |
| *string*.swapcase() | inverts case for all letters in *string* |
| *string*.title()[b][c] | returns "titlecased" version of *string,* that is, all words begin with uppercase, and the rest are lowercase (also see istitle()) |
| *string*.translate(*str, del*="") | translates *string* according to translation table *str*(256 chars), removing those in the *del* string |
| *string*.upper() | converts lowercase letters in *string* to uppercase |
| *string*.zfill (*width*) | returns original *string* left-padded with zeros to a total of *width* characters; intended for numbers, zfill() retains any sign given (less one zero) |

[a] applicable to Unicode strings only in 1.6, but to all string types in 2.0.

[b] not available as a string module function in 1.5.2

[c] not available as a method in JPython 1.1

[d] applicable to Unicode strings only

Using JPython, we will show some examples of methods available for strings:

```
>>> quest = 'what is your favorite color?'
>>> quest.capitalize()
'What is your favorite color?'
>>>
>>> quest.center(40)
'      what is your favorite color?      '
>>>
>>> quest.count('or')
2
>>>
>>> quest.endswith('blue')
0
>>>
>>> quest.endswith('color?')
1
>>>
>>> quest.find('or', 30)
```

```
-1
>>>
>>> quest.find('or', 22)
25
>>
>>> quest.index('or', 10)
16
>>>
>>> ':'.join(quest.split())
'what:is:your:favorite:color?'
>>> quest.replace('favorite color', 'quest')
>>>
'what is your quest?'
>>>
>>> quest.upper()
'WHAT IS YOUR FAVORITE COLOR?'
```

The most complex example shown above is the one with `split()` and `join()`. We first call `split()` on our string, which, without an argument, will break apart our string using spaces as the delimiter. We then take this list of words and call `join()` to merge our words again, but with a new delimiter, the colon. Notice that we used the `split()` method for our string, and the `join()` method for single-character string `':'`.

## Special Features of Strings

### Special or Control Characters

Like most other high-level or scripting languages, a backslash paired with another single character indicates the presence of a "special" character, usually a non-printable character, and that this pair of characters will be substituted by the special character. These are the special characters we discussed above that will not be interpreted if the raw string operator precedes a string containing these characters.

In addition to the well-known characters such as NEWLINE ( `\n` ) and (horizontal) TAB ( `\t` ), specific characters via their ASCII values may be used as well: `\`*`ooo`* or `\x`*`XX`* where *`ooo`* and *`XX`* are their respective octal and hexadecimal ASCII values. Here are the base 10, 8, and 16 representations of 0, 65, and 255:

| | ASCII | ASCII | ASCII |
|---|---|---|---|
| decimal | 0 | 65 | 255 |
| octal | \000 | \101 | \177 |
| hexadecimal | \x00 | \x41 | \xFF |

Special characters, including the backslash-escaped ones, can be stored in Python strings just like regular characters.

Another way that strings in Python are different from those in C is that Python strings are not terminated by the NUL (\000) character (ASCII value 0). NUL characters are just like any of the other special backslash-escaped characters. In fact, not only can NUL characters appear in Python strings, but there can be any number of them in a string, not to mention that they can occur anywhere within the string. They are no more special than any of the other control characters. Table6.7 represents a summary of the escape characters supported by most versions of Python.

**Table 6.7. String Literal Backslash Escape Characters**

| /X | Oct | Dec | Hex | Char | Description |
|---|---|---|---|---|---|
| \0 | 000 | 0 | 0x00 | NUL | Null character |
| \a | 007 | 7 | 0x07 | BEL | Bell |
| \b | 010 | 8 | 0x08 | BS | Backspace |
| \t | 011 | 9 | 0x09 | HT | Horizontal Tab |
| \n | 012 | 10 | 0x0A | LF | Linefeed/Newline |
| \v | 013 | 11 | 0x0B | VT | Vertical Tab |
| \f | 014 | 12 | 0x0C | FF | Form Feed |
| \r | 015 | 13 | 0x0D | CR | Carriage Return |
| \e | 033 | 27 | 0x1B | ESC | Escape |
| \" | 042 | 34 | 0x22 | " | Double quote |
| \' | 047 | 39 | 0x27 | ' | Single quote/apostrophe |
| \\ | 134 | 92 | 0x5C | \ | Backslash |

And as mentioned before, explicit ASCII octal or hexadecimal values can be given, as well as escaping a NEWLINE to continue a statement to the next line. All valid ASCII character values are between 0 and 255 (octal 0177, hexadecimal 0XFF).

| | |
|---|---|
| \OOO | octal value OOO (range is 0000 to 0177) |
| \xXX | 'x' plus hexadecimal value XX (range is 0X00 to 0xFF) |
| \ | escape NEWLINE for statement continuation |

One use of control characters in strings is to serve as delimiters. In database or Internet/Web processing, it is more than likely that most printable characters are allowed as data items, meaning that they would not make good delimiters.

It becomes difficult to ascertain whether or not a character is a delimiter or a data item, and by using a printable character such as a colon (:) as a delimiter, you are limiting the number of allowed characters in your data, which may not be desirable.

One popular solution is to employ seldomly used, non-printable ASCII values as delimiters. These make the perfect delimiters, freeing up the colon and the other printable characters for more important uses.

## Triple Quotes

Although strings can be represented by single or double quote delimitation, it is often difficult to manipulate strings containing special or non-printable characters, especially the NEWLINE character. Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive single or double quotes (used in pairs, naturally):

```
>>> para_str = """this is a long string that is made up of
… several lines and non-printable characters such as
… TAB ( \t ) and they will show up that way when displayed.
… NEWLINEs within the string, whether explicitly given like
… this within the brackets [ \n ], or just a NEWLINE within
… the variable assignment will also show up.
… """
```

Triple quote lets the developer avoid playing quote and escape character games, all the while bringing at least a small chunk of text closer to WYSIWIG (what you see is what you get) format.

An example below shows you what happens when we use the **`print`** statement to display the contents of this string. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code (\n):

```
>>> print para_str
this is a long string that is made up of
several lines and non-printable characters such as
TAB (    ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [
 ], or just a NEWLINE within
the variable assignment will also show up.
```

We introduced the `len()` built-in sequence type function earlier, which, for strings, gives us the total number of characters in a string.

```
>>> len(para_str)
307
```

Upon applying that function to our string, we get a result of 307, which includes the NEWLINE and TAB characters. Another way to look at the string within the interactive interpreter is by just giving the interpreter the name of the object in question. Here, we will see the "internal" representation of the string, without the special characters being converted to printable ones. If that last NEWLINE we looked at above (after the final word "up" and before the closing triple quotes) is still elusive to you, take a look at the way the string is represented internally below. You will observe that the last character of the string is the aforementioned NEWLINE.

```
>>> para_str

'this is a long string that is made up of\012several lines
and non-printable characters such as\012TAB ( \011 ) and
they will show up that way when displayed.\012NEWLINEs
within the string, whether explicitly given like\012this
within the brackets [ \012 ], or just a NEWLINE
within\012the variable assignment will also show up.\012\'
```

## String Immutability

In Section 4.7.2, we discussed how strings are immutable data types, meaning that their values cannot be changed or modified. This means that if you *do* want to update a string, either by taking a substring, concatenating another string on the end, or concatenating the string in question to the end of another string, etc., a new string object must be created for it.

This sounds more complicated than it really is. Since Python manages memory for you, you won't really notice when this occurs. Any time you modify a string or perform any operation that is contrary to immutability, Python will allocate a new string for you. In the following example, Python allocates space for the strings, `'abc'` and `'def'`. But when performing the addition operation to create the string `'abcdef'`, new space is allocated automatically for the new string.

```
>>> 'abc' + 'def'
'abcdef'
```

Assigning values to variables is no different:

```
>>> string = 'abc'
>>> string = string + 'def'
>>> string
'abcdef'
```

In the above example, it looks like we assigned the string `'abc'` to `string`, then appended the string `'def'` to `string`. To the naked eye, strings look mutable. What you cannot see, however, is the fact that a new string was created when the operation `"s + 'def'"` was performed, and that the new object was then assigned back to `s`. The old string of `'abc'` was deallocated.

Once again, we can use the `id()` built-in function to help show us exactly what happened. If you recall, `id()` returns the "identity" of an object. This value is as close to a "memory address" as we can get in Python.

```
>> string = 'abc'
>>>
>>> id(string)
135060856
>>>
>>> string = string + 'def'
>>> id(string)
135057968
```

Note how the identities are different for the string before and after the update. Another test of mutability is to try to modify individual characters or substrings of a string. We will now show how any update of a single character or a slice is not allowed:

```
>>> string
'abcdef'
>>>
>>> string[2] = 'C'
Traceback (innermost last):
 File "<stdin>", line 1, in ?
AttributeError: __setitem__
>>>
>>> string[3:6] = 'DEF'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: __setslice__
```

Both operations result in an error. In order to perform the actions that we want, we will have to create new strings using substrings of the existing string, then assign those new strings back to `string`:

```
>>> string
'abcdef'
>>>
>>> string = string[0:2] + 'C' + string[3:]
>>> string
'abCdef'
```

```
>>>
>>> string[0:3] + 'DEF'
'abCDEF'
>>>
>>> string = string[0:3] + 'DEF'
>>> string
'abCDEF'
```

So for immutable objects like strings, we make the observation that only valid expressions on the left-hand side of an assignment (to the left of the equals sign [ = ]) must be the variable representation of an entire object such as a string, not single characters or substrings. There is no such restriction for the expression on the right-hand side.

## Unicode Support

Unicode string support, introduced to Python in version 1.6, is used to convert between multiple double-byte character formats and encodings, and include as much functionality to manage these strings as possible. With the addition of string methods (see Section 6.6), Python strings are fully-featured to handle a much wider variety of applications requiring Unicode string storage, access, and manipulation. At the time of this writing, the exact Python specifications have not been finalized. We will do our best here to give an overview of native Unicode 3.0 support in Python:

### `unicode()` Built-in Function

The Unicode built-in function should operate in a manner similar to that of the Unicode string operator (`u/U`). It takes a string and returns a Unicode string.

### `encode()` Built-in Methods

The `encode()` built-in methods take a string and return an equivalent encoded string. `encode()` exists as methods for both regular and Unicode strings in 2.0, but only for Unicode strings in 1.6.

### Unicode Type

There is a new Unicode type named `unicode` that is returned when a Unicode string is sent as an argument to `type()`, i.e., `type(u'')`

### Unicode Ordinals

The standard `ord()` built-in function should work the same way. It was enhanced recently to support Unicode objects. The new `unichr()` built-in function returns a Unicode object for character (provided it is a 32-bit value); a `ValueError` exception is raised, otherwise.

### Coercion

Mixed-mode string operations require standard strings be converted to Unicode objects.

### Exceptions

`UnicodeError` is defined in the exceptions module as subclass of `ValueError`. All exceptions related to Unicode encoding/decoding should be subclasses of `UnicodeError`. Also see the string `encode()` method.

**Table 6.8. Unicode Codecs/Encodings**

| *codec* | *Description* |
|---|---|
| utf-8 | 8-bit variable length encoding (default encoding) |
| utf-16 | 16-bit variable length encoding (little/big endian) |
| utf-16-le | utf-16 but explicitly little endian |
| utf-16-be | utf-16 but explicitly big endian |
| ascii | 7-bit ASCII codepage |
| iso-8859-1 | ISO 8859-1 (Latin 1) codepage |
| unicode-escape | (see Python Unicode Constructors for a definition) |
| raw-unicode-escape | (see Python Unicode Constructors for a definition) |
| native | dump of the internal format used by Python |

### RE Engine Unicode-aware

The new regular expression engine should be Unicode aware. See the `re` Code Module sidebar in the next section (6.8).

### String Format Operator

For Python format strings: `'%s'` does `str(u)` for Unicode objects embedded in Python strings, so the output will be `u.encode (<default encoding>)`. If the format string is an Unicode object, all parameters are coerced to Unicode first and then put together and formatted according to the format string. Numbers are first converted to strings and then to Unicode. Python strings are interpreted as Unicode strings using the <default encoding>. Unicode objects are taken as is. All other string formatters should work accordingly. Here is an example:

```
u"%s %s" % (u"abc", "abc") ? u"abc abc"
```

Specific information regarding Python's support of Unicode strings can be found in the `Misc/unicode.txt` of the distribution. The latest version of this document is always available online at:

http://www.starship.python.net/~lemburg/unicode-proposal.txt

For more help and information on Python's Unicode strings, see the Python Unicode Tutorial at:

http://www.reportlab.com/i18n/python_unicode_tutorial.html

## No Characters or Arrays in Python

We mentioned in the previous section that Python does not support a character type. We can also say that C does not support string types explicitly. Instead, strings in C are merely arrays of individual characters. Our third fact is that Python does not have an "array" type as a primitive (although the `array` module exists if you really have to have one). Implementing strings as character arrays is also deemed unnecessary due to the sequential access ability of strings.

In choosing between single characters and strings, Python wisely uses strings as types. It is much easier manipulating the larger entity as a "blob" since most applications operate on strings as a whole rather than individual characters. Applications will convert strings to integers, ask users to input strings, perform regular expression matches on substrings, search files for specific strings, and will even sort a set of strings like names, etc. How often are individual characters operated on, except for searches (i.e., search-and-replace, search-for-delimiter, etc.)? Probably not often as far as most applications are concerned.

However, such functionality should still be available to the Python programmer. Search-and-replacing can be done with regular expressions and the `re` module, searching for and breaking up strings based on delimiters can be accomplished with `split()`, searching for substrings can be accomplished using `find()` and `rfind()`, and just plain old character membership in a string can be verified with the **`in`** and **`not in`** sequence operators.

We are going to quickly revisit the `chr()` and `ord()` built-in functions that convert between ASCII integer values and their equivalent characters, and describe one of the "features" of C that has been lost to Python because characters are not integer types in Python as they are in C.

One feature of C which is lost is the ability to perform numerical calculations directly on characters, i.e., `'A' + 3`. This is allowed in C because both 'A' as a `char` and 3 as an `int` are integers (1-byte and 2/4-bytes, respectively), but would be a type mismatch in Python because 'A' is a string, 3 is a plain integer, and no such addition ( `+` ) operation exists between numeric and string types.

```
>>> 'B'
'B'
>>> 'B' + 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>>
>>> chr('B')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>>
>>> ord('B')
66
>>> ord('B') + 1
67
>>> chr(67)
'C'
>>> chr(ord('B') + 1)
'C'
```

Our failure scenario occurred when we attempted to increase the ASCII value of 'B' by 1 to get 'C' by addition. Rather than 1-byte integer arithmetic, our solution in Python involves using the `chr()` and `ord()` built-in functions.

## Related Modules

Table 6.9 lists the key related modules for strings that are part of the Python standard library.

**Table 6.9. Related Modules for String Types ( -Unix only)**

| *Module* | *Contents* |
|---|---|
| `string` | string manipulation and utility functions |
| `re` | regular expressions: powerful string pattern matching |
| `struct` | convert strings to/from binary data format |
| `c/StringIO` | string buffer object which behaves like a file |
| `crypt` | performs one-way encryption cipher |
| `rotor` | provides multi-platform en/decryption services |

**NOTE**

There are many utility and manipulation functions out there which deal with strings. You may be familiar with some of them if you have programmed in other high-level languages like C, C++, and Java. Python has tried to integrate the most popular functionality into its operators and built-in functions. Nevertheless, they cannot all be integrated into the language. This is where the `string` module comes in. The `string`

module provides a set of constants as well as module functions that provide additional support for strings.

Some of the key functions in the `string` module include: `ato*()`— three functions which convert from strings to three numeric types, `split()`—splits up a string into a list of strings, `join()`— does the reverse of `split()`: merges a list of strings into a single one, and `find()`—searches for substrings.

Refer to the `string` module documentation for more information and usage of string module attributes. Starting in version 1.6 of Python, many of the functions in the `string` module have been implemented as string methods, a new feature of strings which begins the journey of obsoleting this module. We introduced you to these methods in *Section 6.6*.

**NOTE**

Regular expressions (REs) provide advanced pattern matching scheme for strings. Using a separate syntax which describes these patterns, you can effectively use them as "filters" when passing in the text to perform the searches on. These filters allow you to extract the matched patterns as well as perform find-and-replace or divide up strings based on the patterns that you describe.

The `re` module, introduced in Python 1.5, obsoletes the original `regex` and `regsub` modules from earlier releases. It includes a major upgrade in terms of Python's support for regular expressions, adopting the complete Perl syntax for REs. In Python 1.6, the RE engine has been rewritten, for performance improvements as well as support for Unicode strings.

Some of the key functions in the `re` module include: `compile()`—compiles an RE expression into a reusable RE object, `match()`—attempts to match a pattern from the beginning of a string, `search()`—searches for any matching pattern in the string, and `sub()`—performs a search-and-replace of matches. Some of these functions return match objects with which you can access saved group matches (if any were found). All of *Chapter 15* is dedicated to regular expressions.

## Summary of String Highlights

### Consists of Characters Delimited by Quotation Marks

You can think of a string as a Python data type which you can consider as an array or contiguous set of characters between any pair of Python quotation symbols, or quotes. The two most common quote symbols for Python are the single quote, a single forward

apostrophe ( ' ), and the double quotation mark ( " ). The actual string itself consists entirely of those characters in between and not the quote marks themselves.

Having the choice between two different quotation marks is advantageous because it allows one type of quote to serve as a string delimiter while the other can be used as characters within the string without the need for special escape characters. Strings enclosed in single quotes may contain double quotes as characters and vice versa:

```
>>> quote1 = 'George said, "Good day Madam. How are we today?"'
>>> print quote1
George said, "Good day Madam. How are we today?"
>>> quote2 = "Martha replies, 'We are fine, thank you.'"
>>> print quote2
Martha replies, 'We are fine, thank you.'
```

## Python Does not Support a Separate Character Type

Strings are the only literal sequence type, a sequence of characters. However, characters are not a type, so strings are the lowest-level primitive for character storage and manipulation. Most applications tend to deal with strings as a whole and singular entity. To that end, Python provides a good amount of string utilities in the form of operators, built-in functions, and the contents of the string module. However, Python is flexible, allowing access to individual or groups of characters, if desired. Also see Section 6.7.1. Characters are simply strings of length one.

## String Format Operator ( % ) Provides `printf()`-like Functionality

In Section 6.4.1, we highlighted the `printf()`-like string format operator which provides a familiar interface to formatting data for output, whether to the screen or elsewhere.

## Triple Quotes

In Section 6.7.2, we introduced the notion of triple quotes, which are strings that can have special embedded characters like NEWLINEs and TABs. Triple-quoted strings are delimited by pairs of three single (`' ' '`) or double (`"""`) quotation marks.

## Raw Strings Allow for Special Characters to be Taken Verbatim

In Section 6.4.2, we introduced raw strings and discussed how they do not interpret special characters escaped with the backslash. This makes raw strings ideal for situations where strings must be taken verbatim, for example, when describing regular expressions.

### Unlike C strings, Python strings do not Terminate with NUL or '\0'

One of the problems in C is running off the end of your string into memory that does not belong to you. This occurs when strings in C are not properly terminated with the NUL or '\0' character, which has the ASCII value of zero. Along with managing memory for you, Python also removes this little burden or annoyance. Strings in Python do not terminate with NUL, and you do not have to worry about adding them on. Strings consist entirely of the characters that were designated and nothing more.

## Lists

Like strings, lists provide sequential storage through an index offset and access to single or consecutive elements through slices. However, the comparisons usually end there. Strings consist only of characters and are immutable (cannot change individual elements) while lists are flexible container objects which hold an arbitrary number of Python objects. Creating lists is simple; adding to lists is easy, too, as we see in the following examples.

The objects that you can place in a list can include standard types and objects as well as user-defined ones. Lists can contain different types of objects and are more flexible than an array of C structs or Python arrays (available through the external array module) because arrays are restricted to containing objects of a single type. Lists can be populated, empty, sorted, and reversed. Lists can be grown and shrunk. They can be taken apart and put together with other lists. Individual or multiple items can be inserted, updated, or removed at will.

Tuples share many of the same characteristics of lists and although we have a separate section on tuples, many of the examples and list functions are applicable to tuples as well. The key difference is that tuples are immutable, i.e., read-only, so any operators or functions which allow updating lists, such as using the slice operator on the left-hand side of an assignment, will not be valid for tuples.

### How to Create and Assign Lists

Creating lists is as simple as assigning a value to a variable. You handcraft a list (empty or with elements) and perform the assignment. Lists are delimited by surrounding square brackets ( `[ ]` ).

```
>>> aList = [123, 'abc', 4.56, ['inner', 'list'], 7-9j]
>>> anotherList = [None, 'something to see here']
>>> print aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> print anotherList
[None, 'something to see here']
>>> aListThatStartedEmpty = []
>>> print aListThatStartedEmpty
```

```
[]
```

## How to Access Values in Lists

Slicing works similar to strings; use the square bracket slice operator ( `[ ]` ) along with the index or indices.

```
>>> aList[0]
123
>>> aList[1:4]
['abc', 4.56, ['inner', 'list']]
>>> aList[:3]
[123, 'abc', 4.56]
>>> aList[3][1]
'list'
```

## How to Update Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the `append()` method:

```
>>> aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> aList[2]
4.56
>>> aList[2] = 'float replacer'
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>>
>>> anotherList.append("hi, i'm new here")
>>> print anotherList
[None, 'something to see here', "hi, i'm new here"]
>>> aListThatStartedEmpty.append('not empty anymore')
>>> print aListThatStartedEmpty
['not empty anymore']
```

## How to Remove List Elements and Lists

To remove a list element, you can use either the **`del`** statement if you know exactly which element(s) you are deleting or the `remove()` method if you do not know.

```
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
```

```
>>> del aList[1]
>>> aList
[123, 'float replacer', ['inner', 'list'], (7-9j)]
>>> aList.remove(123)
>>> aList
['float replacer', ['inner', 'list'], (7-9j)]
```

You can also use the `pop()` method to remove and return a specific object from a list.

Normally, removing an entire list is not something application programmers do. Rather, they tend to let it go out of scope (i.e., program termination, function call completion, etc.) and be garbage-collected, but if they do want to explicitly remove an entire list, use the **`del`** statement:

```
del aList
```

# Operators

## Standard Type Operators

In Chapter 4, we introduced a number of operators that apply to most objects, including the standard types. We will take a look at how some of those apply to lists.

```
>>> list1 = [ 'abc', 123 ]
>>> list2 = [ 'xyz', 789 ]
>>> list3 = [ 'abc', 123 ]
>>> list1 < list2
1
>>> list2 < list3
0
>>> (list2 > list3) and (list1 == list3)
1
```

When using the value comparison operators, comparing numbers and strings is straightforward, but not so much for lists, however. List comparisons are somewhat tricky, but logical. The comparison operators use the same algorithm as the `cmp()` built-in function. The algorithm basically works like this: the elements of both lists are compared until there is a determination of a winner. For example, in our example above, the output of `'abc'` versus `'xyz'` is determined immediately, with `'abc' < 'xyz'`, resulting in `list1 < list2` and `list2 >= list3`. Tuple comparisons are performed in the same manner as lists.

## Sequence Type Operators

### Slices ([ ] and [ : ])

Slicing with lists is very similar to strings, but rather than using individual characters or substrings, slices of lists pull out an object or a group of objects which are elements of the list operated on. Focusing specifically on lists, we make the following definitions:

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
```

Slicing operators obey the same rules regarding positive and negative indexes, starting and ending indexes, as well as missing indexes, which default to the beginning or to the end of a sequence.

```
>>> num_list[1]
-1.23
>>>
>>> num_list[1:]
[-1.23, -2, 619000.0]
>>>
>>> num_list[2:-1]
[-2]
>>>
>>> str_list[2]
'over'
>>> str_list[:2]
['jack', 'jumped']
>>>
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>> mixup_list[1]
[1, 'x']
```

Unlike strings, an element of a list might also be a sequence, implying that you can perform all the sequences operations or execute any sequence built-in functions on that element. In the example below, we show that not only can we take a slice of a slice, but we can also change it, and even to an object of a different type. You will also notice the similarity to multi-dimensional arrays.

```
>>> mixup_list[1][1]
'x'
>>> mixup_list[1][1] = -64.875
>>> mixup_list
[4.0, [1, -64.875], 'beef', (-1.9+6j)]
```

Here is another example using num_list:

```
>>> num_list
[43, -1.23, -2, 6.19e5]
>>>
>>> num_list[2:4] = [ 16.0, -49 ]
>>>
>>> num_list
[43, -1.23, 16.0, -49]
>>>
>>> num_list[0] = [65535L, 2e30, 76.45-1.3j]
>>>
>>> num_list
[[65535L, 2e+30, (76.45-1.3j)], -1.23, 16.0, -49]
```

Notice how, in the last example, we replaced only a single element of the list, but we replaced it with a list. So as you can tell, removing, adding, and replacing things in lists are pretty free-form. Keep in mind that in order to splice elements of a list into another list, you have to make sure that the left-hand side of the assignment operator (=) is a slice, not just a single element.

### Membership (in, not in)

With strings, the membership operator determined whether a single character is a member of a string. With lists (and tuples), we can check whether an object is a member of a list (or tuple).

```
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> 'beef' in mixup_list
1
>>>
>>> 'x' in mixup_list
0
>>>
>>> 'x' in mixup_list[1]
1
>>> num_list
[[65535L, 2e+030, (76.45-1.3j)], -1.23, 16.0, -49]
>>>
>>> -49 in num_list
1
>>>
>>> 34 in num_list
0
>>>
>>> [65535L, 2e+030, (76.45-1.3j)] in num_list
1
```

Note how `'x'` is *not* a member of `mixup_list`. That is because `'x'` itself is not actually a member of `mixup_list`. Rather, it is a member of `mixup_uplist[1]`, which itself is a list. The membership operator is applicable in the same manner for tuples.

### Concatenation (+)

The concatenation operator allows us to join multiple lists together. Note in the examples below that there is a restriction of concatenating like objects. In other words, you can concatenate only objects of the same type. You cannot concatenate two different types even if both are sequences.

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
>>>
>>> num_list + mixup_list
[43, -1.23, -2, 619000.0, 4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> str_list + num_list
['jack', 'jumped', 'over', 'candlestick', 'park', 43, -1.23, -2,
619000.0]
```

As we will discover in Section 6.13, starting in Python 1.5.2, you can use the `extend()` method in place of the concatenation operator to append the contents of a list to another. Using `extend()` is advantageous over concatenation because it actually appends the elements of the new list to the original, rather than creating a new list from scratch like `+` does. `extend()` is also the method used by the new augmented assignment or in-place concatenation operator (`+=`) which debuted in Python 2.0.

We would also like to point out that the concatenation operator *does not* facilitate adding individual elements to a list. The upcoming example illustrates a case where attempting to add a new item to the list results in failure.

```
>>> num_list + 'new item'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

This example fails because we had different types to the left and right of the concatenation operator. A combination of (list `+` string) is not valid. Obviously, our intention was to add the `'new item'` string to the list, but we did not go about it the proper way. Fortunately, we have a solution:

Use the `append()` list built-in method (we will formally introduce `append()` and all other built-in methods in Section 6.13):

```
>>> num_list.append('new item')
```

#### Repetition (*)

Use of the repetition operator may make more sense with strings, but as a sequence type, lists and tuples can also benefit from this operation, if needed:

```
>>> num_list * 2
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0]
>>>
>>> num_list * 3
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0, 43, -1.23, -2,
619000.0]
```

A new augmented assignment in-place repetition operator was also added to Python 2.0.

### List Type Operators

There are currently no special list-only operators in Python. Lists can be used with most object and sequence operators. In addition, list objects have their own methods.

## Built-in Functions

### Standard Type Functions

#### cmp()

In Section 4.6.1, we introduced the `cmp()` built-in function with examples of comparing numbers and strings. But how would `cmp()` work with other objects such as lists and tuples, which can contain not only numbers and strings, but other objects like lists, tuples, dictionaries, and even user-created objects?

```
>>> list1, list2 = [123, 'xyz'], [456, 'abc']
>>> cmp(list1, list2)
-1
>>>
>>> cmp(list2, list1)
1
>>> list3 = list2 + [789]
>>> list3
[456, 'abc', 789]
```

```
>>>
>>> cmp(list2, list3)
-1
```

Compares are straightforward if we are comparing two objects of the same type. For numbers and strings, the direct values are compared, which is trivial. For sequence types, comparisons are somewhat more complex, but similar in manner. Python tries its best to make a fair comparison when one cannot be made, i.e., when there is no relationship between the objects or when types do not even have compare functions, then all bets are off as far as obtaining a "logical" decision.

Before such a drastic state is arrived at, more safe-and-sane ways to determine an inequality are attempted. How does the algorithm start? As we mentioned briefly above, elements of lists are iterated over. If these elements are of the same type, the standard compare for that type is performed. As soon as an inequality is determined in an element compare, that result becomes the result of the list compare. Again, these element compares are for elements of the same type. As we explained earlier, when the objects are different, performing an accurate or true comparison becomes a risky proposition.

When we compare `list1` with `list2,` both lists are iterated over. The first true comparison takes place between the first elements of both lists, i.e., `123` vs. `456.` Since `123 < 456, list1` is deemed "smaller."

If both values are the same, then iteration through the sequences continues until either a mismatch is found, or the end of the shorter sequence is reached. In the latter case, the sequence with more elements is deemed "greater." That is the reason why we arrived above at `list2 < list3.` Tuples are compared using the same algorithm. We leave this section with a summary of the algorithm highlights:

Compare elements of both lists.

If elements are of the same type, perform the compare and return the result.

If elements are different types, check to see if they are numbers.

If numbers, perform numeric coercion if necessary and compare.

If either element is a number, then the other element is "larger" (numbers are "smallest").

Otherwise, types are sorted alphabetically by name.

If we reached the end of one of the lists, the longer list is "larger."

If we exhaust both lists and share the same data, the result is a tie, meaning that 0 is returned.

## Sequence Type Functions

### len()

For strings, `len()` gives the total length of the string, as in the number of characters. For lists (and tuples), it will not surprise you that `len()` returns the number of elements in the list (or tuple). Container objects found within count as a single item. Our examples below use some of the lists already defined above in previous sections.

```
>>> len(num_list)
4
>>>
>>> len(num_list*2)
8
>>>
>>> len(str_list[:4])
4
>>>
>>> len(str_list[:-1])
4
>>>
>>> len(mixup_list+num_list)
8
```

### max() and min()

`max()` and `min()` did not have a significant amount of usage for strings since all they did was to find the "largest" and "smallest" characters (lexicographically) in the string. For lists (and tuples), their functionality is more defined. Given a list of like objects, i.e., numbers or strings only, `max()` and `min()` could come in quite handy. Again, the quality of return values diminishes as mixed objects come into play. However, more often than not, you will be using these functions in a situation where they will provide the results you are seeking. We present a few examples using some of our earlier-defined lists.

```
>>> max(str_list)
'park'
>>>
>>> max(num_list)
[65535L, 2e+30, (76.45-1.3j)]
>>> max(mixup_list)
'beef'
>>> min(mixup_list)
(-1.9+6j)
>>>
>>> min(str_list)
'candlestick'
>>>
>>> min(num_list)
-49
```

### `list()` and `tuple()`

The `list()` and `tuple()` methods take sequence types and convert them to lists and tuples, respectively. Although strings are also sequence types, they are not commonly used with `list()` and `tuple()`. These built-in functions are used more often to convert from one type to the other., i.e., when you have a tuple that you need to make a list (so that you can modify its elements) and vice versa.

```
>>> aList = [ 'tao', 93, 99, 'time' ]
>>> aTuple = tuple(aList)
>>> print aList
['tao', 93, 99, 'time']
>>>
>>> print aTuple
('tao', 93, 99, 'time')
>>>
>>> back2aList = list(aTuple)
>>> back2aList
['tao', 93, 99, 'time']
>>> back2aList == aList
1
>>> back2aList is aList
0
```

Neither `list()` nor `tuple()` performs true *conversions* (also see Section 6.1.2). In other words, the list you passed to `tuple()` does not turn into a list, and the tuple you give to `list()` does not really become a list. Instead, these built-in functions create a new object of the destination type and populate it with the same elements as the original sequence. In the last two examples above, although the data set for both lists is the same (hence satisfying `==`), neither variable points to the same list (thus failing `is`).

### List Type Built-in Functions

There are currently no special list-only built-in functions in Python. Lists can be used with most object and sequence built-in functions. In addition, list objects have their own methods.

## List Type Built-in Methods

Lists in Python have *methods*. We will go over methods more formally in an introduction to object-oriented programming in Chapter 13, but for now, think of methods as functions or procedures that apply only to specific objects. So the methods described in this section behave just like built-in functions except that they operate only on lists. Since these functions involve the mutability (or updating) of lists, none of them is applicable for tuples.

You may recall our earlier discussion of accessing object attributes using the dotted attribute notation: *object.attribute*. List methods are no different, using *list.method([arguments])*. We use the dotted notation to access the attribute (here it is a function), then use the function operators ( ( ) ) in a functional notation to invoke the methods.

Types that have methods generally have an attribute called *object*.__methods__ which name all the methods that are supported by that type. In our case for lists, *list*.__methods__ serves this purpose:

```
>>> [].__methods__
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

Table6.10 shows all the current methods currently available for lists. Of these methods, extend() and pop() made their debut in Python 1.5.2. Some examples of using various list methods are shown below.

**Table 6.10. List Type Built-in Methods**

| *List Method* | *Operation* |
|---|---|
| *list*.append(*obj*) | appends object *obj* to *list* |
| *list*.count(*obj*) | returns count of how many times *obj* occurs in *list* |
| *list*.extend(*seq*) [a] | appends the contents of *seq* to *list* |
| *list*.index(*obj*) | returns the lowest index in *list* that obj appears |
| *list*.insert(*index, obj*) | inserts object *obj* into *list* at offset *index* |
| *list*.pop(*obj=list*[-1]) [a] | removes and returns last object or *obj* from *list* |
| *list*.remove(*obj*) | removes object *obj* from *list* |
| *list*.reverse() | reverses objects of *list* in place |
| *list*.sort([*func*]) | sorts objects of list, use compare *func* if given |

[a] new as of Python 1.5.2

```
>>> music_media = [45]
>>> music_media
[45]
>>>
>>> music_media.insert(0, 'compact disc')
>>> music_media
['compact disc', 45]
>>>
>>> music_media.append('long playing record')
>>> music_media
['compact disc', 45, 'long playing record']
>>>
>>> music_media.insert(2, '8-track tape')
```

```
>>> music_media
['compact disc', 45, '8-track tape', 'long playing record']
```

In the preceeding example, we initiated a list with a single element, then checked the list as we either inserted elements within the list, or appended new items at the end. Let's now determine if elements are in a list as well as how to find out the location of where items are in a list. We do this by using the **in** operator and `index()` method.

```
>>> 'cassette' in music_media
0
>>> 'compact disc' in music_media
1
>>> music_media.index(45)
1
>>> music_media.index('8-track tape')
2
>>> music_media.index('cassette')
Traceback (innermost last):
 File "<interactive input>", line 0, in ?
ValueError: list.index(x): x not in list
```

Oops! What happened in that last example? Well, it looks like using `index()` to check if items are in a list is not a good idea, because we get an error. It would be safer to check using the membership operator **in** (or **not in**) first, and then using `index()` to find the element's location. We can put the last few calls to `index()` in a single for loop like this:

```
for eachMediaType in (45, '8-track tape', 'cassette'):
    if eachMediaType in music_media:
        print music_media.index(eachMediaType)
```

This solution helps us avoid the error we encountered above because `index()` is not called unless the object was found in the list. We will find out later how we can take charge if the error occurs, instead of bombing out as we did above.

We will now test drive `sort()` and `reverse(),` methods that will sort and reverse the elements of a list, respectively.

```
>>> music_media
['compact disc', 45, '8-track tape', 'long playing record']
>>> music_media.sort()
>>> music_media
[45, '8-track tape', 'compact disc', 'long playing record']
>>> music_media.reverse()
>>> music_media
```

```
['long playing record', 'compact disc', '8-track tape', 4
```

One caveat about the `sort()` and `reverse()` methods is that these will perform their operation on a list in place, meaning that the contents of the existing list will be changed. There is no return value from either of these methods.

Oh, and if you are an algorithm connoisseur, the default sorting algorithm employed by the `sort()` method is a randomized version of QuickSort. We defer all other explanation to the portion of the source code where you can find out more information on the sorting algorithm (`Objects/listobject.c`).

The new `extend()` method will take the contents of one list and append its elements to another list:

```
>>> new_media = ['24/96 digital audio disc', 'DVD Audio
disc', 'Super Audio CD']
>>> music_media.extend(new_media)
>>> music_media
['long playing record', 'compact disc', '8-track tape',
45, '24/96 digital audio disc', 'DVD Audio disc', 'Super
Audio CD']
```

The argument to `extend()` can be any sequence object starting in Python 1.6—the sequence is converted to a list by performing the equivalent to `list()` and then its contents appended to the original list. In 1.5.2, the argument was required to be a list.

`pop()` will either return the last or requested item from a list and return it to the caller. We will see the new `pop()` method in Section 6.14.1 as well as in the Exercises.

## Special Features of Lists

### Creating Other DataStructures Using Lists

Because of their container and mutable features, lists are fairly flexible and it is not very difficult to build other kinds of data structures using lists. Two that we can come up with rather quickly are stacks and queues.

The two code samples in this section use the `pop()` method which became reality in Python 1.5.2. If you are using an older system, this function is easily duplicated in Python. (See Exercise 6-17.)

#### Stack

A stack is a last-in-first-out (LIFO) data structure which works similar to a cafeteria dining plate spring-loading mechanism. Consider the plates as objects. The first object off the stack is the last one you put in. Every new object gets "stacked" on top of the newest objects. To "push" an item on a stack is the terminology used to mean you are adding onto a stack. Likewise, to remove an element, you "pop" it off the stack. The following example shows a menu-driven program which implements a simple stack used to store strings:

**Example 6.2. Using Lists as a Stack (`stack.py`)**

This simple script uses lists as a stack to store and retrieve strings entered through this menu-driven text application using only the `append()` and `pop()` list methods.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  stack = []
004 4
005 5  def pushit():
006 6      stack.append(raw_input('Enter new string: '))
007 7
008 8  def popit():
009 9      if len(stack) == 0:
010 10         print 'Cannot pop from an empty stack!'
011 11     else: <$nopage>
012 12         print 'Removed [', stack.pop(), ']'
013 13
014 14 def viewstack():
015 15     print str(stack)
016 16
017 17 def showmenu():
018 18     prompt = """
019 19 p(U)sh
020 20 p(O)p
021 21 (V)iew
022 22 (Q)uit
023 23
024 24 Enter choice: """
025 25
026 26     done = 0
027 27     while not done:
028 28
029 29         chosen = 0
030 30         while not chosen:
031 31            try: <$nopage>
032 32                 choice = raw_input(prompt)[0]
033 33            except (EOFError, KeyboardInterrupt):
034 34                 choice = 'q'
035 35            print 'nYou picked: [%s]' % choice
036 36            if choice not in 'uovq':
037 37                print 'invalid option, try again'
038 38             else:
039 39               chosen = 1
040 40
```

```
041 41         if choice == 'q': done = 1
042 42         if choice == 'u': pushit()
043 43         if choice == 'o': popit()
044 44         if choice == 'v': viewstack()
045 45
046 46     if __name__ == '__main__':
047 47     showmenu()
048  <$nopage>
```

**Lines 1–3**

In addition to the Unix startup line, we take this opportunity to clear the stack (a list).

**Lines 5–6**

The `pushit()` function adds an element (a string prompted from the user) to the stack.

**Lines 8–12**

The `popit()` function removes an element from the stack (the more recent one). An error occurs when trying to remove an element from an empty stack. In this case, a warning is sent back to the user.

**Lines 14–15**

The `viewstack()` function displays a printable string representation of the list.

**Lines 17–44**

The entire menu-driven application is controlled from the `showmenu()` function. Here, the user is prompted with the menu options. Once the user makes a valid choice, the proper function is called. We have not covered exceptions and **`try-except`** statement in detail yet, but basically that section of the code allows a user to type ^D (EOF, which generates an `EOFError`) or ^C (interrupt to quit, which generates a `KeyboardInterrupt` error), both of which will be processed by our script in the same manner as if the user had typed the `'q'` to quit the application. This is one place where the exception-handling feature of Python comes in extremely handy.

**Lines 46–47**

This part of the code starts up the program if invoked directly. If this script was imported as a module, only the functions and variables would have been defined, but the menu would not show up. For more information regarding line 46 and the `__name__` variable, see Section 3.4.1.

Here is a sample execution of our script:

```
% stack.py
```

```
p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: u

You picked: [u]
Enter new string: Python

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: u

You picked: [u]
Enter new string: is

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: u

You picked: [u]
Enter new string: cool!

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: v

You picked: [v]
['Python', 'is', 'cool!']

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o

You picked: [o]
Removed [ cool! ]

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o
```

```
You picked: [o]
Removed [ is ]

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o

You picked: [o]
Removed [ Python ]

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o

You picked: [o]
Cannot pop from an empty stack!

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: ^D

You picked: [q]
```

### Queue

A queue is a first-in-first-out (FIFO) data structure which works like a single-file supermarket or bank teller line. The first person in line is the first one served (and hopefully the first one to exit). New elements join by being "enqueued" at the end of the line, and elements are removed from the front by being "dequeued." The following code shows how, with a little modification from our stack script, we can implement a simple queue using lists.

#### Example 6.3. Using Lists as a Queue (`queue.py`)

This simple script uses lists as a queue to store and retrieve strings entered through this menu-driven text application, using only the `append()` and `pop()` list methods.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  queue = []
004 4
```

```
005 5  def enQ():
006 6      queue.append(raw_input('Enter new string: '))
007 7
008 8  def deQ():
009 9      if len(queue) == 0:
010 10        print 'Cannot dequeue from empty queue!'
011 11     else: <$nopage>
012 12        print 'Removed [', queue.pop(0), ']'
013 13
014 14  def viewQ():
015 15      print str(queue)
016 16
017 17  def showmenu():
018 18      prompt = """
019 19  (E)nqueue
020 20  (D)equeue
021 21  (V)iew
022 22  (Q)uit
023 23
024 24  Enter choice: """
025 25
026 26     done = 0
027 27     while not done:
028 28
029 29          chosen = 0
030 30         while not chosen:
031 31               try: <$nopage>
032 32                   choice = raw_input(prompt)[0]
033 33              except (EOFError, KeyboardInterrupt):
034 34                   choice = 'q'
035 35               print '\nYou picked: [%s]' % choice
036 36               if choice not in 'devq':
037 37                 print 'invalid option, try again'
038 38               else: <$nopage>
039 39                   chosen = 1
040 40
041 41         if choice == 'q': done = 1
042 42         if choice == 'e': enQ()
043 43         if choice == 'd': deQ()
044 44         if choice == 'v': viewQ()
045 45
046 46     if __name__ == '__main__':
047 47         showmenu()
048  <$nopage>
```

Because of the similarities of this script with the `stack.py` script, we will describe only in detail the lines which have changed significantly:

### Lines 5–6

The `enQ()` function works exactly like `pushit()`, only the name has been changed.

### Lines 8–12

The key difference between the two scripts lies here. The `deQ()` function, rather than taking the most recent item as `popitem()` did, takes the oldest item on the list, the first element.

We present some output here as well:

```
% queue.py
(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: e

You picked: [e]
Enter new queue element: Bring out

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: e

You picked: [e]
Enter new queue element: your dead!

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: v

You picked: [v]
['Bring out', 'your dead!']

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: d

You picked: [d]
Removed [ Bring out ]

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: d

You picked: [d]
```

```
     Removed [ your dead! ]

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: d

You picked: [d]
Cannot dequeue from empty queue!

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: ^D
You picked: [q]
```

### Subclassing from "Lists"

Earlier in this text, we described how types are not classes in Python, so you cannot derive subclasses from them (see the Core Note in Section 4.2). As a proxy, the Python standard library includes two modules containing class wrappers around two types, lists and dictionaries, from which you can subclass. These are the `UserList` and `UserDict` modules. Once you are familiar with classes, you can take these already-implemented classes to create your own subclasses from lists and dictionaries and add whatever functionality you wish. These modules are part of the Python standard library. See Section 6.18 for more information.

## Tuples

Tuples are another container type extremely similar in nature to lists. The only visible difference between tuples and lists is that tuples use parentheses and lists use square brackets. Functionally. there is a more significant difference, and that is the fact that tuples are immutable.

Our usual *modus operandi* is to present the operators and built-in functions for the more general objects, followed by those for sequences and conclude with those applicable only for tuples, but because tuples share so many characteristics with lists, we would be duplicating much of our description from the previous section. Rather than providing much repeated information, we will differentiate tuples from lists as they apply to each set of operators and functionality, then discuss immutability and other features unique to tuples.

### How to Create and Assign Tuples

Creating and assigning lists are practically identical to lists, with the exception of empty tuples. These require a trailing comma ( , ) enclosed in the tuple delimiting parentheses ( ( ) ).

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> anotherTuple = (None, 'something to see here')
>>> print aTuple
 (123, 'abc', 4.56, ['inner', 'tuple'], (7-9j))
>>> print anotherTuple
 (None, 'something to see here')
>>> emptiestPossibleTuple = (None,)
>>> print emptiestPossibleTuple
(None,)
```

## How to Access Values in Tuples

Slicing works similar to lists: Use the square bracket slice operator ( [ ] ) along with the index or indices.

```
>>> aTuple>>> aList[1:4]
('abc', 4.56, ['inner', 'tuple'])
>>> aTuple[:3]
(123, 'abc', 4.56)
>>> aTuple[3][1]
'tuple'
```

## How to Update Tuples

Like numbers and strings, tuples are immutable which means you cannot update them or change values of tuple elements. In Sections 6.2 and 6.3.2, we were able to take portions of an existing string to create a new string. The same applies for tuples.

```
>>> aTuple = aTuple[0], aTuple[1], aTuple[-1]
>>> aTuple
(123, 'abc', (7-9j))
>>> tup1 = (12, 34.56)
>>> tup2 = ('abc', 'xyz')
>>> tup3 = tup1 + tup2
>>> tup3
(12, 34.56, 'abc', 'xyz')
```

## How to Remove Tuple Elements and Tuples

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire list, just use the **del** statement:

```
del aTuple
```

# Tuple Operators and Built-in Functions

## Standard and Sequence TypeOperators and Built-in Functions

Object and sequence operators and built-in functions act the exact same way toward tuples as they do with lists. You can still take slices of tuples, concatenate and make multiple copies of tuples, validate membership, and compare tuples:

### Creation, Repetition, Concatenation

```
>>> t = (['xyz', 123], 23, -103.4)
>>> t
(['xyz', 123], 23, -103.4)
>>> t * 2
(['xyz', 123], 23, -103.4, ['xyz', 123], 23, -103.4)
>>> t = t + ('free', 'easy')
>>> t
(['xyz', 123], 23, -103.4, 'free', 'easy')
```

### Membership, Slicing

```
>>> 23 in t
1
>>> 123 in t
0
>>> t[0][1]
123
>>> t[1:]
 (23, -103.4, 'free', 'easy')
```

### Built-in Functions

```
>>> str(t)
(['xyz', 123], 23, -103.4, 'free', 'easy')
>>> len(t)
5
```

```
>>> max(t)
'free'
>>> min(t)
-103.4
>>> cmp(t, (['xyz', 123], 23, -103.4, 'free', 'easy'))
0
>>> list(t)
[['xyz', 123], 23, -103.4, 'free', 'easy']
```

#### Operators

```
>>> (4, 2) < (3, 5)
0
>>> (2, 4) < (3, -1)
1
>>> (2, 4) == (3, -1)
0
>>> (2, 4) == (2, 4)
1
```

### Tuple Type Operators and Built-in Functions and Methods

Like lists, tuples have no operators or built-in functions for themselves. All of the list methods described in the previous section were related to a list object's mutability, i.e., sorting, replacing, appending, etc. Since tuples are immutable, those methods are rendered superfluous, thus unimplemented.

## Special Features of Tuples

### How are Tuples Affected by Immutability?

Okay, we have been throwing around this word "immutable" in many parts of the text. Aside from its computer science definition and implications, what is the bottom line as far as applications are concerned? What are all the consequences of an immutable data type?

Of the three standard types which are immutable—numbers, strings, and tuples—tuples are the most affected. A data type that is immutable simply means that once an object is defined, its value cannot be updated, unless, of course, a completely new object is allocated. The impact on numbers and strings is not as great since they are scalar types, and when the sole value they represent is changed, that is the intended effect, and access occurs as desired. The story is different with tuples, however.

Because tuples are a container type, it is often desired to change a single or multiple elements of that container. Unfortunately, this is not possible. Slice operators cannot

show up on the left-hand side of an assignment. Recall this is no different for strings, and that slice access is used for read access only.

Immutability does not necessarily mean bad news. One bright spot is that if we pass in data to an API with which we are not familiar, we can be certain that our data will not be changed by the function called. Also, if we receive a tuple as a return argument from a function that we would like to manipulate, we can use the `list()` built-in function to turn it into a mutable list.

## Tuples Are Not Quite So "Immutable"

Although tuples are defined as immutable, this does not take away from their flexibility. Tuples are not quite as immutable as we made them out to be. What do we mean by that? Tuples have certain behavioral characteristics that make them seem not as immutable as we had first advertised.

For example, we can join strings together to form a larger string. Similarly, there is nothing wrong with putting tuples together to form a larger tuple, so concatenation works. This process does not involve changing the smaller individual tuples in any way. All we are doing is joining their elements together. Some examples are presented here:

```
>>> s = 'first'
>>> s = s + ' second'
>>> s
'first second'
>>>
>>> t = ('third', 'fourth')
>>> t
('third', 'fourth')
>>>
>>> t = t + ('fifth', 'sixth')
>>> t
('third', 'fourth', 'fifth', 'sixth')
```

The same concept applies for repetition. Repetition is just concatenation of multiple copies of the same elements. In addition, we mentioned in the previous section that one can turn a tuple into a mutable list with a simple function call. Our final feature may surprise you the most. You can "modify" certain tuple elements. Whoa. What does that mean?

Although tuple objects themselves are immutable, this fact does not preclude tuples from containing mutable objects which *can* be changed.

```
>>> t = (['xyz', 123], 23, -103.4)
>>> t
(['xyz', 123], 23, -103.4)
>>> t[0][1]
```

```
123
>>> t[0][1] = ['abc', 'def']
>> t
(['xyz', ['abc', 'def']], 23, -103.4)
```

In the above example, although `t` is a tuple, we managed to "change" it by replacing an item in the first tuple element (a list). We replaced `t[0][1],` formerly an integer, with a list `['abc', 'def'].` Although we modified only a mutable object, in some ways, we also "modified" our tuple.

## No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples:

```
>>> 'abc', -4.24e93, 18+6.6j, 'xyz'
('abc', -4.24e+093, (18+6.6j), 'xyz')
>>>
>>> x, y = 1, 2
>>> x, y
(1, 2)
```

Any function returning multiple objects (also no enclosing symbols) is a tuple. Note that enclosing symbols change a set of multiple objects returned to a single container object. For example:

```
def foo1():
    :
    return
                                                    obj1, obj2, obj3
def foo2():
    :
    return [obj1, obj2, obj3]
def foo3():
    :
    return (obj1, obj2, obj3)
```

In the above examples, `foo1()` calls for the return of three objects, which come back as a tuple of three objects, `foo2()` returns a single object, a list containing three objects, and `foo3()` returns the same thing as `foo1().` The only difference is that the tuple grouping is explicit.

Explicit grouping of parentheses for expressions or tuple creation is always recommended to avoid unpleasant side effects:

```
>>> 4, 2 < 3, 5      # int, comparison, int
(4, 1, 5)
>>> (4, 2) < (3, 5)  # tuple comparison
0
```

In the first example, the less than ( `<` ) operator took precedence over the comma delimiter intended for the tuples on each side of the less than sign. The result of the evaluation of `2 < 3` became the second element of a tuple. Properly enclosing the tuples enables the desired result.

## Single Element Tuples

Ever try to create a tuple with a single element? You tried it with lists, and it worked, but then you tried and tried with tuples, but cannot seem to be able to do it.

```
>>> ['abc']
['abc']
>>> type(['abc'])    # a list
<type 'list'>
>>>
>>> [123]
[123]
>>> type([123])      # also a list
<type 'list'>
>>>
>>> ('xyz')
'xyz'
>>> type(('xyz'))    # a string, not a tuple
<type 'string'>
>>>
>>> (456)
456
>>> type((456))      # an int, not a tuple
<type 'int'>
```

It probably does not help your case that the parentheses are also overloaded as the expression grouping operator. Parentheses around a single element take on that binding role rather than as a delimiter for tuples. The workaround is to place a trailing comma (,) after the first element to indicate that this is a tuple and not a grouping.

```
>>> ('xyz',)
('xyz',)
>>> (456,)
```

```
(456,)
```

**NOTE**

One of the questions in the Python FAQ (6.15) asks, "Why are there separate tuple and list data types?" That question can also be rephrased as, "Do we really need two sequence types?" One reason why having lists and tuples is a good thing occurs in situations where having one is more advantageous than having the other.

One case in favor of an immutable data type is if you were manipulating sensitive data and were passing a mutable object to an unknown function (perhaps an API that you didn't even write!). As the engineer developing your piece of the software, you would definitely feel a lot more secure if you knew that the function you were calling could not alter the data.

An argument for a mutable data type is where you are managing dynamic data sets. You need to be able to create them on the fly, slowly or arbitrarily adding to them, or from time to time, deleting individual elements. This is definitely a case where the data type must be mutable. The good news is that with the `list()` and `tuple()` built-in conversion functions, you can convert from one type to the other relatively painlessly

`list()` and `tuple()` are functions which allow you to create a tuple from a list and vice versa. When you have a tuple and want a list because you need to update its objects, the `list()` function suddenly becomes your best buddy. When you have a list and want to pass it into a function, perhaps an API, and you do not want anyone to mess with the data, the `tuple()` function comes in quite useful.

## Related Modules

Table6.11 lists the key related modules for sequence types. This list includes the `array` module to which we briefly alluded earlier. These are similar to lists except for the restriction that all elements must be of the same type. The copy module (see optional Section 6.19 below) performs shallow and deep copies of objects. The `operator` module, in addition to the functional equivalents to numeric operators, also contains the same four sequence types. The `types` module is a reference of type objects representing all types which Python supports, including sequence types. Finally, the `UserList` module contains a full class implementation of a list object. Because Python types cannot be subclassed, this module allows users to obtain a class that is list-like in nature, and derive new classes or functionality. If you are unfamiliar with object-oriented programming, we highly recommend reading Chapter 13.

**Table 6.11. Related Modules for Sequence Types**

| *Module* | *Contents* |
|---|---|
| `array` | features the `array` restricted mutable sequence type which requires all of its elements to be of the same type |

| | |
|---|---|
| copy | provides functionality to perform shallow and deep copies of objects (see 6.19 below for more information) |
| operator | contains sequence operators available as function calls, i.e. operator.concat(*m*, *n*) is equivalent to the concatenation (*m* + *n*) for sequences *m* and *n*. |
| types | contains type objects for all supported Python types |
| UserList | wraps a list object (including operators and methods) into a class which can be used for derivation (also see Section 6.18) |

## *Shallow and Deep Copies

Earlier in Section 3.5, we described how object assignments are simply object references. This means that when you create an object, then assign that object to another variable, Python does not copy the object. Instead, it copies only a *reference* to the object. For example:

```
>>> aList = [[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>> anotherList = aList
>>> aList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>>
>>> anotherList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
```

Above, a list of two elements is created and its reference assigned to aList. When aList is assigned to anotherList, the contents of the list reference by aList are not copied when anotherList is created. Rather, anotherList "copies" the reference from aList, not the data. We can confirm this by taking a look at the identities of the objects that both references point to:

```
>>> id(aList)
1191872
>>> id(anotherList)
1191872
```

A *shallow copy* of an object is defined to be a newly-created object of the same type as the original object whose contents are references to the elements in the original object. In other words, the copied object itself is new, but the contents are not. Shallow copies of sequence objects may be taken one of two ways: (1) taking a complete slice using the slice operator, or (2) using the copy() function of the copy module, as indicated in the example below:

```
>>> thirdList = aList[:]
>>> thirdList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>> id(thirdList)
```

```
1192232
>>>
>>> import copy
>>> fourthList = copy.copy(aList)
>>> fourthList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>> id(fourthList)
1192304
```

The `thirdList` list is created using the slice operator to take an entire slice (both starting and ending indices are absent). We also present the new object's identity to confirm its disassociation with the original object. Likewise for the creation of the `fourthList` list. This time, we use the `copy.copy()` function to perform the same feat. However, the elements of these lists are still only references to the original object's elements.

```
>>> id(aList[0]), id(aList[1]), id(aList[2])
(1064072, 1191920, 1191896)
>>> id(thirdList[0]), id(thirdList[1]), id(thirdList[2])
(1064072, 1191920, 1191896)
>>> id(fourthList[0]), id(fourthList[1]), id(fourthList[2])
(1064072, 1191920, 1191896)
```

We pull the identities of these objects to confirm our hypothesis. In order to obtain a full or *deep copy* of the object—creating a new container but containing references to completely new copies (references) of the element in the original object—we need to use the `copy.deepcopy()` function.

```
>>> lastList = copy.deepcopy(aList)
>>> lastList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>> id(lastList)
1193248
>>> id(lastList[0]), id(lastList[1]), id(lastList[2])
(1192280, 1193128, 1193104)
```

There are a few notes and caveats to making copies to keep in mind. The first is that non-container types (i.e., numbers, strings, and other "atomic" objects like code, type, and xrange objects) are not copied. Shallow copies of sequences are all done using complete slices. Mapping types, which will be covered in Chapter 8, are copied using the dictionary copy method. Finally, deep copies of tuples are not made if they contain only atomic objects. If we changed each of the small lists in the larger list above to all tuples, we would have performed only a shallow copy, even though we requested a deep copy.

**NOTE**

The shallow and deep copy operations that we just described are found in the `copy` module. There are really only two functions to use from this module: `copy()`—creates shallow copy, and `deepcopy()`—creates a deep copy.

Sequence types provide various mechanisms for ordered storage of data. Strings are a general medium for carrying data, whether it be displayed to a user, stored on a disk, transmitted across the network, or be a singular container for multiple sources of information. Lists and tuples provide container storage that allows for simple manipulation and access of multiple objects, whether they by Python data types or user-defined objects. Individual or groups of elements may be accessed as slices via sequentially-ordered index offsets. Together, these data types provide flexible and easy-to-use storage tools in your Python development environment. We conclude this chapter with a summary of operators, built-in functions and methods for sequence types given as Table6.12.

**Table 6.12. Sequence Type Operators, Built-in Functions and Methods**

| *Operator, built-in function or method* | *String* | *List* | *Tuple* |
|---|---|---|---|
| `[]` (list creation) | | • | |
| `()` | | | • |
| `''` | • | | |
| `append()` | | • | |
| `capitalize()` | • | | |
| `center()` | • | | |
| `chr()` | • | | |
| `cmp()` | • | • | • |
| `count()` | • | • | |
| `encode()` | • | | |
| `endswith()` | • | | |
| `expandtabs()` | • | | |
| `extend()` | | • | |
| `find()` | • | | |
| `hex()` | • | | |
| `index()` | • | • | |
| `insert()` | • | • | |
| `isdecimal()` | • | | |
| `isdigit()` | • | | |
| `islower()` | • | | |
| `isnumeric()` | • | | |
| `isspace()` | • | | |
| `istitle()` | • | | |
| `isupper()` | • | | |

| | | | |
|---|---|---|---|
| join() | • | | |
| len() | • | • | • |
| list() | • | • | • |
| ljust() | • | | |
| lower() | • | | |
| lstrip() | • | | |
| max() | • | • | • |
| min() | • | • | • |
| oct() | • | | |
| ord() | • | | |
| pop() | | • | |
| raw_input() | • | | |
| remove() | | • | |
| replace() | • | | |
| repr() | • | • | |
| reverse() | | • | |
| rfind() | • | | |
| rindex() | • | | |
| rjust() | • | | |
| rstrip() | • | | |
| sort() | | • | |
| split() | • | | |
| splitlines() | • | | |
| startswith() | • | | |
| str() | • | • | • |
| strip() | • | | |
| swapcase() | • | | |
| split() | • | | |
| title() | • | | |
| tuple() | • | • | • |
| type() | • | • | • |
| upper() | • | | |
| zfill() | • | | |
| . (attributes) | • | • | |
| [] (slice) | • | • | • |
| [:] | • | • | • |
| * | • | • | • |
| % | • | | |
| + | • | • | • |
| **in** | • | • | • |
| **not in** | • | • | • |

## Exercises

**1:**

*Strings.* Are there any string methods or functions in the string module that will help me determine if a string is part of a larger string?

**2:**

*String Identifiers.* Modify the `idcheck.py` script in Example 6-1 such that it will determine the validity of identifiers of length 1 as well as be able to detect if an identifier is a keyword. For the latter part of the exercise, you may use the `keyword` module (specifically the `keyword.kwlist` list) to aid in your cause.

**3:**

*Sorting.*

(a) Enter a list of numbers and sort the values in largest-to-smallest order.

(b) Do the same thing, but for strings and in reverse alphabetical (largest-to-smallest lexicographic) order.

**4:**

*Arithmetic.* Update your solution to the test score exercise in the previous chapter such that the test scores are entered into a list. Your code should also be able to come up with an average score. See Exercises 2-9 and 5-3.

**5:**

*Strings.*

(a) Display a string one character-at-a-time forward *and* backward as well.

(b) Determine if two strings match (without using comparison operators or the `cmp()` built-in function) by scanning each string. EXTRA CREDIT: Add case-insensitivity to your solution.

(c) Determine if a string is palindromic (the same backwards as it is for wards). EXTRA CREDIT: add code to suppress symbols and white space if you want to process anything other than strict palindromes.

(d) Take a string and append a backwards copy of that string, making a palindrome.

**6:**

*Strings.* Create the equivalent to `string.strip()`: Take a string and remove all

leading and trailing whitespace. (Use of `string.*strip()` defeats the purpose of this exercise.)

**7:**

*Debugging.* Take a look at the code we present in Example 6.4 (`buggy.py`).

(a) Study the code and describe what this program does. Add a comment to every place you see a comment sign ( `#` ). Run the program.

(b) This problem has a big bug in it. It fails on inputs of 6, 12, 20, 30, etc., not to mention any even number in general. What is wrong with this program?

(c) Fix the bug in (b).

**Example 6.4. buggy program(`buggy.py`)**

This is the program listing for Exercise 6-7. You will determine what this program does, add comments where you see "#"s, determine what is wrong with it, and provide a fix for it.

```
 <$nopage>
001 1 #!/usr/bin/env python
002 2
003 3 #
004 4 import string
005 5
006 6 #
007 7 num_str = raw_input('Enter a number: ')
008 8
009 9 #
010 10 num_num = string.atoi(num_str)
011 11
012 12 #
013 13 fac_list = range(1, num_num+1)
014 14 print "BEFORE:", 'fac_list'
015 15
016 16 #
017 17 i = 0
018 18
019 19 #
020 20 while i < len(fac_list):
021 21
022 22     #
023 23     if num_num % fac_list[i] == 0:
024 24         del fac_list[i]
025 25
026 26     #
027 27     i = i + 1
028 28
029 29  #
030 30 print "AFTER:", 'fac_list'
```

```
031  <$nopage>
```

**8:**

*Lists.* Given an integer value, return a string with the equivalent English text of each digit. For example, an input of 89 results in "eight nine" being returned. EXTRA CREDIT: return English text with proper usage, i.e., "eighty-nine." For this part of the exercise, restrict values to be between zero and a thousand.

**9:**

*Conversion.* Create a sister function to your solution for Exercise 6-13 to take the total number of minutes and return the same time interval in hours and minutes, maximizing on the total number of hours.

**10:**

*Strings.* Create a function that will return another string similar to the input string, but with its case inverted. For example, input of "Mr. Ed" will result in "mR. eD" as the output string.

**11:**

Conversion.

(a) Create a program that will convert from an integer to an Internet Protocol (IP) address in the four octet format of WWW.XXX.YYY.ZZZ.

(b) Update your program to be able to do the vice versa of the above.

**12:**

*Strings.*

(a) Create a function called `findchr()`, with the following declaration:

```
def findchr(string, char)
```

`findchr()` will look for character char in *`string`* and return the index of the first occurrence of *`char`*, or -1 if that *`char`* is not part of *`string`*. You cannot use `string.*find()` or `string.*index()` functions or methods.

(b) Create another function called `rfindchr()` which will find the last occurrence of a character in a string. Naturally this works similarly to `findchr()` but it starts its search from the end of the input string.

(c) Create a third function called `subchar()` with the following declaration:

```
def subchr(string, origchar, newchar)
```

`subchr()` is similar to `findchr()` except that whenever *`origchar`* is found, it is replaced by *`newchar`*. The modified string is the return value.

**13:**

*Strings.* The `string` module contains three functions, `atoi(), atol(),` and `atof(),` that convert strings to integers, long integers, and floating point numbers, respectively. As of Python 1.5, the Python built-in functions `int(), long(),` and `float()` can also perform the same tasks, in addition to `complex()` which can turn a string into a complex number. (Prior to 1.5 however, those built-in functions converted only between numeric types.)

An `atoc()` was never implemented in the `string` module, so that is your task here. `atoc()` takes a single string as input, a string representation of a complex number, i.e., `'-1.23e+4-5.67j',` and returns the equivalent complex number object with the given value. You cannot use `eval(),` but `complex()` is available. However, you can use only `complex()` with the following restricted syntax: `complex(`*`real, imag`*`)` where `real` and `imag` are floating point values. See Table 6.4 for more information regarding the use of `complex().`

**14:**

**Random numbers.* Design a "rock, paper, scissors" game, sometimes called "Rochambeau," a game you may have played as a kid. If you don't know the rules, they are: at the same time, both you and your opponent have to pick from one of the following: rock, paper, or scissors using specified hand motions. The winner is determined by these rules, which form somewhat of a fun paradox: (a) the paper covers the rock, (b) the rock breaks the scissors, (c) the scissors cut the paper. In your computerized version, the user enters his/her guess, the computer randomly chooses, and your program should indicate a winner or draw/tie. NOTE: the most algorithmic solutions use the fewest number of **`if`** statements.

**15:**

Conversion.

(a) Given a pair of dates in some recognizable standard format such as MM/DD/YY or DD/MM/YY, determine the total number of days that fall between both dates.

(b) Given a person's birth date, determine the total number of days that person has

been alive, including all leap days.

(c) Armed with the same information from part (b) above, determine the number of days remaining until that person's next birthday.

**16:**

*Matrices.* Process the addition and multiplication of a pair of M by N matrices.

**17:**

Methods. Implement a function called `my pop()`, which is similar to the `list pop()` method. Take a list as input, remove the last object from the list and return it.

# Chapter 7. Dictionaries

In this chapter, we focus on Python's single mapping type, dictionaries. We present the various operators and built-in functions which can be used with dictionaries. We conclude this chapter by introducing some of the standard library modules which deal with dictionaries.

## Introduction to Dictionaries

The last standard type to add to our repertoire is the dictionary, the sole mapping type in Python. A dictionary is mutable and is another container type that can store any number of Python objects, including other container types. What makes dictionaries different from sequence type containers like lists and tuples is the way the data is stored and accessed.

Sequence types use numeric keys only (numbered sequentially as indexed offsets from the beginning of the sequence). Mapping types may use most other object types as keys, strings being the most common. Unlike sequence type keys, mapping keys are often, if not directly, associated with the data value that is stored. But because we are no longer using "sequentially-ordered" keys with mapping types, we are left with an unordered collection of data. As it turns out, this does not hinder our use because mapping types do not require a numeric value to index into a container to obtain the desired item. With a key, you are "mapped" directly to your value, hence the term "mapping type." The most common data structure that maps keys with associated values are *hash tables.*

**NOTE**

Sequence types use sequentially-ordered numeric keys as index offsets to store your data in an array format. The index number usually has nothing to do with the data value that is being stored. There should also be a way to store data based on another, associated value such as a string. We do this all the time in everyday living. You file people's phone numbers in your address book based on last name, you add events to your calendar or appointment book based on date and time, etc. For each of these examples, an associated value to a data item was your key.

Hash tables are a data structure that does exactly what we described. They store each piece of data, called a value, based on an associated data item, called a key. Together, these are known as key-value pairs. The hash table algorithm takes your key, performs an operation on it, called a hash function, and based on the result of the calculation, chooses where in the data structure to store your value. Where any one particular value is stored

depends on what its key is. Because of this randomness, there is no ordering of the values in the hash table. You have an unordered collection of data.

The only kind of ordering you can obtain is by key. You can request a dictionary's keys, which is returned to you as a list. From there, you can call the list's `sort()` method to order that data set. This is only one type of ordering you can perform on your keys. In any case, once you have determined that the set of keys is "sorted" to your satisfaction, their associated values may be retrieved from the dictionary. Hash tables generally provide good performance because lookups occur fairly quickly once you have a key. For a sequential access data structure, you must march down to the correct index location and then retrieve the value. Naturally, performance is based on the type of hash function used.

Python dictionaries are implemented as resizeable hash tables. If you are familiar with Perl, then we can say that dictionaries are similar to Perl's associative arrays or hashes.

We will now take a closer look at Python dictionaries. The syntax of a dictionary entry is `key:value`. Also, dictionary entries are enclosed in braces ( { } ).

## How to Create and Assign Dictionaries

Creating dictionaries simply involves assigning a dictionary to a variable, regardless of whether the dictionary has elements or not:

```
>>> dict1 = {}
>>> dict2 = {'name': 'earth', 'port': 80}
>>> dict1, dict2
({}, {'port': 80, 'name': 'earth'})
```

## How to Access Values in Dictionaries

To access dictionary elements, you use the familiar square brackets along with the key to obtain its value:

```
>>> dict2['name']
'earth'
>>>
>>> print 'host %s is running on port %d' % \
…     (dict2['name'], dict2['port'])
host earth is running on port 80
```

Dictionary `dict1` is empty while `dict2` has two data items. The keys in `dict2` are `'name'` and `'port',` and their associated value items are `'earth'` and `80,` respectively. Access to the value is through the key, as you can see from the explicit access to the `'name'` key.

If we attempt to access a data item with a key which is not part of the dictionary, we get an error:

```
>>> dict2['server']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: server
```

In this example, we tried to access a value with the key `'server'` which, as you know, does not exist from the code above. The best way to check if a dictionary has a specific key is to use the dictionary's `has_key()` method. We will introduce all of a dictionary's methods below. The Boolean `has_key()` method will return a 1 if a dictionary has that key and 0 otherwise.

```
>>> dict2.has_key('server')
0
>>> dict2.has_key('name')
1
>>> dict2['name']
'earth'
```

Once the `has_key()` method has given the okay, meaning that a key exists, then you can access it without having to worry about getting the `KeyError,` similar to what happened above. Let us take a look at another dictionary example, using keys other than strings:

```
>>> dict3 = {}
>>> dict3[1] = 'abc'
>>> dict3['1'] = 3.14159
>>> dict3[3.2] = 'xyz'
>>> dict3
{3.2: 'xyz', 1: 'abc', '1': 3.14159}
```

Rather than adding each key-value pair individually, we could have also entered all the data for `dict3` at the same time:

```
dict3 = { 3.2: 'xyz', 1: 'abc', '1': 3.14159 }
```

Creating the dictionary with a set key-value pair can be accomplished if all the data items are known in advance (obviously). The goal of the examples using `dict3` is to illustrate the variety of keys that you can use. If we were to pose the question of whether a key for a particular value should be allowed to change, you would probably say, "No." Right?

Not allowing keys to change during execution makes sense if you think of it this way: Let us say that you created a dictionary element with a key and value. Somehow during execution of your program, the key changed, perhaps due to an altered variable. When you went to retrieve that data value again with the original key, you got a `KeyError` (since the key changed), and you had no idea how to obtain your value now because the key had somehow been altered. Because of this reason, keys must be immutable, so numbers and strings are fine, but lists and other dictionaries are not. (See Section 7.5.2 for why keys must be immutable.)

## How to Update Dictionaries

You can update a dictionary by adding a new entry or element (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry (see below for more details on removing an entry).

```
>>> dict2['name'] = 'venus'          # update existing entry
>>> dict2['port'] = 6969             # update existing entry
>>> dict2['arch'] = 'sunos5'         # add new entry
>>>
>>> print 'host %(name)s is running on port %(port)d' % dict2
host venus is running on port 6969
```

If the key does exist, then its previous value will be overridden by its new value. The **`print`** statement above illustrates an alternative way of using the string format operator ( `%` ), specific to dictionaries. Using the dictionary argument, you can shorten the `print` request somewhat because naming of the dictionary occurs only once, as opposed to occurring for each element using a tuple argument.

You may also add the contents of an entire dictionary to another dictionary by using the `update()` built-in method. We will introduce this methods later on in this chapter in Section 7.4.

## How to Remove Dictionary Elements and Dictionaries

Removing an entire dictionary is not a typical operation. Generally, you either remove individual dictionary elements or clear the entire contents of a dictionary. However, if you really want to "remove" an entire dictionary, use the **`del`** statement (introduced in Section 3.5.6). Here are some deletion examples for dictionaries and dictionary elements:

```
del dict1['name']          # remove entry with key 'name'

dict1.clear()              # remove all entries in dict1

del dict1                  # delete entire dictionary
```

**NOTE**

You may recall that there are two ways to delete an entry from a list, using the **del** statement or using the *list.* remove() method. Then you must be wondering, why do lists have a remove entry method but not dictionaries? One simple answer is that to remove a element from a list is a two-step effort. You must first find the index (a.k.a. the key) where the data item is located and then call the **del** statement. The remove() method was written to perform both steps, leaving the programmer with a single step. With dictionaries, you already have the key; there is no need to perform a lookup. You just call **del** once. Creating a dictionary method to remove an entry will provide you with a functional interface.

## Operators

Dictionaries do not support sequence operations such as concatenation and repetition, although an update() built-in method exists that populates one dictionary with the contents of another. Dictionaries do not have a "membership" operator either, but the has_key() built-in method basically performs the same task.

### Standard Type Operators

Dictionaries will work with all of the standard type operators. These were introduced in Chapter 4, but we will present some examples of how to use them with dictionaries here:

```
>>> dict4 = { 'abc': 123 }
>>> dict5 = { 'abc': 456 }
>>> dict6 = { 'abc': 123, 98.6: 37 }
>>> dict7 = { 'xyz': 123 }
>>> dict4 < dict5
1
>>> (dict4 < dict6) and (dict4 < dict7)
1
>>> (dict5 < dict6) and (dict5 < dict7)
1
>>> dict6 < dict7
0
```

How are all these comparisons performed? Like lists and tuples, the process is a bit more complex than it is for numbers and strings. The algorithm is detailed below in Section 7.3.1.

### Dictionary Key-lookup Operator ( [ ] )

The only operator specific to dictionaries is the key-lookup operator, which works very similar to the single element slice operator for sequence types.

For sequence types, an index offset is the sole argument or subscript to access a single element of a sequence. For a dictionary, lookups are by key, so that is the argument rather than an index. The key-lookup operator is used for both assigning values to and retrieving values from a dictionary:

```
dict[k] = v     # set value 'v' in dictionary with key 'k'
dict[k]         # lookup value in dictionary with key 'k'
```

## Built-in Functions

### Standard Type Functions [type(), str(), and cmp()]

The `type()` built-in function, when operated on a dictionary, reveals an object of the dictionary type. The `str()` built-in function will produce a printable string representation of a dictionary. These are fairly straightfoward.

In each of the last three chapters, we showed how the `cmp()` built-in function worked with numbers, strings, lists, and tuples. So how about dictionaries? Comparisons of dictionaries are based on an algorithm which starts with sizes first, then keys, and finally values. In our example below, we create two dictionaries and compare them, then slowly modify the dictionaries to show how these changes affect their comparisons:

```
>>> dict1 = {}
>>> dict2 = { 'host': 'earth', 'port': 80 }
>>> cmp(dict1, dict2)
-1
>>> dict1['host'] = 'earth'
>>> cmp(dict1, dict2)
-1
```

In the first comparison, `dict1` is deemed smaller because `dict2` has more elements (2 items vs. 0 items). After adding one element to `dict1`, it is still smaller (2 vs. 1), even if the item added is also in `dict2`.

```
>>> dict1['port'] = 8080
```

```
>>> cmp(dict1, dict2)
1
>>> dict1['port'] = 80
>>> cmp(dict1, dict2)
0
```

After we add the second element to `dict1`, both dictionaries have the same size, so their keys are then compared. At this juncture, both sets of keys match, so comparison proceeds to checking their values. The values for the `'host'` keys are the same, but when we get to the `'port'` key, `dict2` is deemed larger because its value is greater than that of `dict1`'s `'port'` key (8080 vs. 80). When resetting `dict2`'s `'port'` key to the same value as `dict1`'s `'port'` key, then both dictionaries form equals: They have the same size, their keys match, and so do their values, hence the reason that 0 is returned by `cmp()`.

```
>>> dict1['prot'] = 'tcp'
>>> cmp(dict1, dict2)
1
>>> dict2['prot'] = 'udp'
>>> cmp(dict1, dict2)
-1
```

As soon as an element is added to one of the dictionaries, it immediately becomes the "larger one," as in this case with `dict1`. Adding another key-value pair to `dict2` can tip the scales again, as both dictionaries' sizes match and comparison progresses to checking keys and values.

```
>>> cdict = { 'fruits':1 }
>>> ddict = { 'fruits':1 }
>>> cmp(cdict, ddict)
0
>>> cdict['oranges'] = 0
>>> ddict['apples'] = 0
>>> cmp(cdict, ddict)
14
```

Our final example reminds as that `cmp()` may return values other than -1, 0, or 1. The algorithm pursues comparisons in the zzfollowing order:

**(1) Compares Dictionary Sizes**

If the dictionary lengths are different, then for `cmp(dict1, dict2)`, `cmp()` will return a positive number if *`dict1`* is longer and a negative number of *`dict2`* is longer. In other words, the dictionary with more keys is greater, i.e.,

```
len(dict1) > len(dict2) ? dict1 > dict2
```

### (2) Compares Dictionary Keys

If both dictionaries are the same size, then their keys are compared; the order in which the keys are checked is the same order as returned by the `keys()` method. (It is important to note here that keys which are the same will map to the same locations in the hash table. This keeps key-checking consistent.) At the point where keys from both do not match, they are directly compared and `cmp()` will return a positive number if the first differing key for *`dict1`* is greater than the first differing key of *`dict2`*.

### (3) Compares Dictionary Values

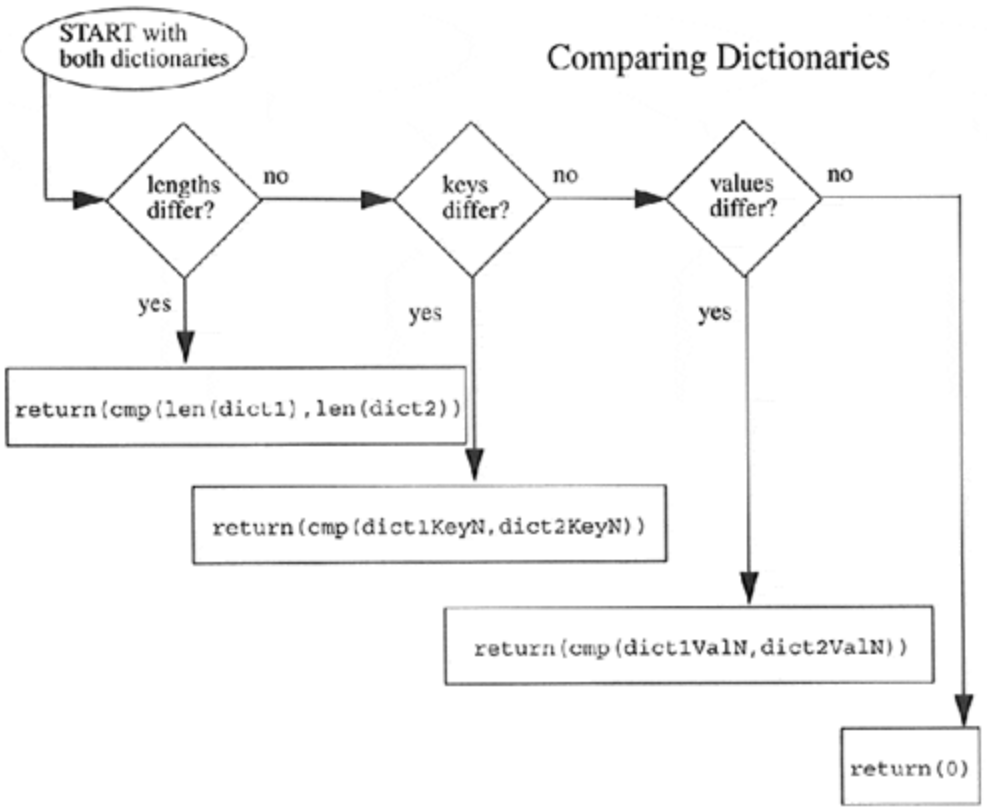
If both dictionary lengths are the same and the keys match exactly, the values for each key in both dictionaries are compared. Once the first key with non-matching values is found, those values are compared directly. Then `cmp()` will return a positive number if, using the same key, the value in *`dict1`* is greater than that of the value in *`dict2`*.

### (4) Exact Match

If we have reached this point, i.e., the dictionaries have the same length, the same keys, and the same values for each key, then the dictionaries are an exact match and 0 is returned.

Figure7-1 illustrates the dictionary compare algorithm we just outlined above.

**Figure 7-1. How Dictionaries are Compared**




### Mapping Type Function [ `len()` ]

Similar to the sequence type built-in function, the mapping type `len()` built-in returns the total number of items, that is, key-value pairs, in a dictionary:

```
>>> dict2 = { 'name': 'earth', 'port': 80 }
>>> dict2
{'port': 80, 'name': 'earth'}
>>> len(dict2)
2
```

We mentioned earlier that dictionary items are unordered. We can see that above, when referencing `dict2`, the items are listed in reverse order from which they were entered into the dictionary.

## Built-in Methods

Table 7.1 lists the methods for dictionary objects. The `clear()`, `copy()`, `get()`, and `update()` methods were added recently in Python 1.5. `setdefault()` was introduced in 2.0.

| Table 7.1. Dictionary Type Methods | |
|---|---|
| ***dictionary method*** | ***Operation*** |
| `dict.clear`[a] `()` | removes all elements of dictionary *dict* |
| `dict.copy`[a]`()` | returns a (shallow [b] ) copy of dictionary *dict* |
| `dict.get(key, default=None)`[a] | for key `key`, returns value or *default* if *key* not in dictionary (note that *default*'s default is `None`) |
| `dict.has_key(key)` | returns 1 if *key* in dictionary *dict*, 0 otherwise |
| `dict.items()` | returns a list of *dict*'s (key, value) tuple pairs |
| `dict.keys()` | returns list of dictionary *dict*'s keys |
| `dict.setdefault key, default=None)`[c] | similar to `get()`, but will set `dict[key]=default` if `key` is not already in `dict` |
| `dict.update(dict2)`[a] | adds dictionary *dict2*'s key-values pairs to *dict* |
| `dict.values()` | returns list of dictionary *dict*'s values |

[a] new as of Python 1.5

[b] more information regarding shallow and deep copies can be found in Section 6.19

[c] new as of Python 2.0

Below, we showcase some of the more common dictionary methods:

```
>>> dict2 = { 'name': 'earth', 'port': 80 }
>>> dict2.has_key('name')
1
>>>
>>> dict2['name']
'earth'
>>>
>>> dict2.has_key('number')
0
```

The `has_key()` method is Boolean, indicating whether the given key is valid for the dictionary the method is operating on. Attempting to access a non-existent key will result in an exception (`KeyError`) as we saw at the beginning of this chapter in Section 7.1. Mapping types do not support the `in` and **`not in`** operators as sequences do, so `has_key()` is our best bet.

Other useful dictionary methods focus entirely on their keys and values. These are `keys()`, which returns a list of the dictionary's keys, `values()`, which returns a list of the dictionary's values, and `items()`, which returns a list of (key, value) tuple pairs. These are useful for when you wish to iterate through a dictionary's keys or values, albeit in no particular order.

```
>>> dict2.keys()
```

```
['port', 'name']
>>>
>>> dict2.values()
[80, 'earth']
>>>
>>> dict2.items()
[('port', 80), ('name', 'earth')]
>>>
>>> for eachKey in dict2.keys():
…    print 'dict2 key', eachKey, 'has value',
dict2[eachKey]
…
dict2 key port has value 80
dict2 key name has value earth
```

The `keys()` method is fairly useful when used in conjunction with a **`for`** loop to retrieve a dictionary's values as it returns a list of a dictionary's keys. However, because its items are unordered, imposing some type of order is usually desired. Below, we present the same the loop, but sort the keys (using the list's `sort()` method) before retrieval.

```
>>> dict2Keys = dict2.keys()
>>> dict2Keys.sort()
>>> for eachKey in dict2Keys:
…       print 'dict2 key', eachKey, 'has value',
dict2[eachKey]
…
dict2 key name has value earth
dict2 key port has value 80
```

The `update()` method can be used to add the contents of one directory to another. Any existing entries with duplicate keys will be overridden by the new incoming entries. Non-existent ones will be added. All entries in a dictionary can be removed with the `clear()` method.

```
>>> dict2= { 'host':'earth', 'port':80 }
>>> dict3= { 'host':'venus', 'server':'http' }
>>> dict2.update(dict3)
>>> dict2
{'server': 'http', 'port': 80, 'host': 'venus'}
>>> dict3.clear()
>>> dict3
{}
```

The `copy()` method simply returns a copy of a dictionary. Note that this is a shallow copy only. Again, see Section 6.19 regarding shallow and deep copies. Finally, the `get()` method is similar to using the key-lookup operator ( [ ] ), but allows you to provide a

default value returned if a key does not exist. If a key does not exist and a default value is not given, then `None` is returned. This is a more flexible option than just using key-lookup because you do not have to worry about an exception being raised if a key does not exist.

```
>>> dict4 = dict2.copy()
>>> dict4
{'server': 'http', 'port': 80, 'host': 'venus'}
>>> dict4.get('host')
'venus'
>>> dict4.get('xxx')
>>> type(dict4.get('xxx'))
<type 'None'>
>>> dict4.get('xxx', 'no such key')
'no such key'
```

Python 2.0 introduces a new dictionary built-in method, `setdefault()`, which is intended on making code shorter by collapsing a common idiom: you want to check if a dictionary has a key. If it does, you want its value. If the dictionary does not have the key you are seeking, you want to set a default value and then return it. That is precisely what `setdefault()` does:

```
>>> myDict = { 'host': 'earth', 'port': 80 }
>>> myDict.keys()
['host', 'port']
>>> myDict.items()
[('host', 'earth'), ('port', 80)]
>>> myDict.setdefault('port', 8080)
80
>>> myDict.setdefault('prot', 'tcp')
'tcp'
>>> myDict.items()
[('prot', 'tcp'), ('host', 'earth'), ('port', 80)]
```

For more information, take a look at the "What's New in 2.0" online document. The URL is available in the Online Resources section of the Appendix and on the CD-ROM.

## Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, i.e., from standard objects to user-defined objects. However, the same cannot be said of keys.

### More Than One Entry per Key Not Allowed

One rule is that you are constrained to having only one entry per key. In other words, multiple values per the same key are not allowed. (Container objects such as lists, tuples,

and other dictionaries are fine.) When key *collisions* are detected (meaning duplicate keys encountered during assignment), the last assignment wins.

```
>>> dict1 = {' foo':789, 'foo': 'xyz'}
>>> dict1
{'foo': 'xyz'}
>>>
>>> dict1['foo'] = 123
>>> dict1
{'foo': 123}
```

Rather than producing an error, Python does not check for key collisions because that would involve taking up memory for each key-value pair assigned. In the above example where the key `'foo'` is given twice on the same line, Python applies the key-value pairs from left to right. The value `789` may have been set at first, but is quickly replaced by the string `'xyz'`. When assigning a value to a non-existent key, the key is created for the dictionary and value added, but if the key does exist (a collision), then its current value is replaced. In the above example, the value for the key `'foo'` is replaced twice; in the final assignment, `'xyz'` is replaced by `123`.

## Keys Must Be Immutable

As we mentioned earlier in Section 7.1, most Python objects can serve as keys—only mutable types such as lists and dictionaries are disallowed. In other words, types that compare by value rather than by identity cannot be used as dictionary keys. A `TypeError` will occur if a mutable type is given as the key:

```
>>> dict[[3]] = 14
Traceback (innermost last):
  File "<stdin>," line 1, in ?
TypeError: unhashable type
```

Why must keys be immutable? The hash function used by the interpreter to calculate where to store your data is based on the value of your key. If the key was a mutable object, its value could be changed. If a key changes, the hash function will map to a different place to store the data. If that was the case, then the hash function could never reliably store or retrieve the associated value. Immutable keys were chosen for the very fact that their values cannot change. (Also see the Python FAQ question 6.18.)

We know that numbers and strings are allowed as keys, but what about tuples? We know they are immutable, but in Section 6.17.2, we hinted that they might not be as immutable as they can be. The clearest example of that was when we modified a list object which was one of our tuple elements. To allow tuples as valid keys, one more restriction must

be enacted: Tuples are valid keys only if they only contain immutable arguments like numbers and strings.

We conclude this chapter on dictionaries by presenting a program (`userpw.py` as in Example 7.1), which manages user name and passwords in a mock login entry database system. This script accepts new users given that they provide a login name and a password. Once an "account" has been set up, an existing user can return as long as they give their login and correct password. New users cannot create an entry with an existing login name.

### Example 7.1. Dictionary Example (`userpw.py`)

This application manages a set of users who join the system with a login name and a password. Once established, existing users can return as long as they remember their login and password. New users cannot create an entry with someone else's login name.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  db = {}
004 4
005 5  def newuser():
006 6      prompt = 'login desired: '
007 7      while 1:
008 8          name = raw_input(prompt)
009 9          if db.has_key(name):
010 10             prompt = 'name taken, try another: '
011 11             continue <$nopage>
012 12         else: <$nopage>
013 13             break <$nopage>
014 14     pwd = raw_input('passwd: ')
015 15     db[name] = pwd
016 16
017 17 def olduser():
018 18     name = raw_input('login: ')
019 19     pwd = raw_input('passwd: ')
020 20     passwd = db.get(name)
021 21     if passwd == pwd:
022 22         pass <$nopage>
023 23     else: <$nopage>
024 24         print 'login incorrect'
025 25         return <$nopage>
026 26
027 27     print 'welcome back', name
028 28
029 29 def showmenu():
030 30     prompt = """
031 31 (N)ew User Login
032 32 (E)xisting User Login
033 33 (Q)uit
034 34
035 35 Enter choice: """
036 36
```

```
037 37     done = 0
038 38     while not done:
039 39
040 40         chosen  = 0
041 41         while not chosen:
042 42             try: <$nopage>
043 43                 choice = raw_input(prompt)[0]
044 44             except (EOFError, KeyboardInterrupt):
045 45                 choice = 'q'
046 46             print '\nYou picked: [%s]' % choice
047 47             if choice not in 'neq':
048 48                 print 'invalid option, try again'
049 49             else: <$nopage>
050 50                 chosen = 1
051 51
052 52         if choice == 'q': done = 1
053 53         if choice == 'n': newuser()
054 54         if choice == 'e': olduser()
055 55
056 56 if __name__ == '__main__':
057 57     showmenu()
058  <$nopage>
```

**Lines 1 – 3**

After the UNIX-startup line, we initialize the program with an empty user database. Because we are not storing the data anywhere, a new user database is created every time this program is executed.

**Lines 5 – 15**

The `newuser()` function is the code that serves new users. It checks to see if a name has already been taken, and once a new name is verified, the user is prompted for his or her password (no encryption exists in our simple program), and his or her password is stored in the dictionary with his or her user name as the key.

**Lines 17 – 27**

The `olduser()` function handles returning users. If a user returns with the correct login and password, a welcome message is issued. Otherwise, the user is notified of an invalid login and returned to the menu. We do not want an infinite loop here to prompt for the correct password because the user may have inadvertently entered the incorrect menu option.

**Lines 29 – 54**

The real controller of this script is the `showmenu()` function. The user is presented with a friendly menu. The prompt string is given using triple quotes because it takes place over multiple lines and is easier to manage on multiple lines than on a single line with embedded `'\n'` symbols. Once the menu is displayed, it waits for valid input from the user and chooses which mode of operation to follow based on the menu choice. The **try-**

**except** statements we describe here are the same as for the `stack.py` and `queue.py` examples from the last chapter (see Section 6.14.1).

### Lines 56 – 57

Here is the familiar code which will only call `showmenu()` to start the application if the script was involved directly (not imported).

Here is a sample execution of our script:

```
% userpw.py

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: n

You picked: [n]
login desired: king arthur
passwd: grail

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: e

You picked: [e]
login: sir knight
passwd: flesh wound
login incorrect

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: e

You picked: [e]
login: king arthur
passwd: grail
welcome back king arthur

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: ^D
You picked: [q]
```

## Exercises

**1:**

*Dictionary Methods.* What dictionary method would we use to combine two dictionaries together?

**2:**

*Dictionary Keys.* We know that dictionary values can be arbitrary Python objects, but what about the keys? Try using different types of objects as the key other than numbers or strings. What worked for you and what didn't? As for the failures, why do you think they didn't succeed?

**3:**

*Dictionary and List Methods.*

(a) Create a dictionary and display its keys alphabetically.

(b) Now display both the keys and values sorted in alphabetical order by the key.

(c) Same as part (b), but sorted in alphabetical order by the value. (Note: this generally has no practical purpose in dictionaries or hash tables in general because most access and ordering [if any] is based on the keys. This is merely an exercise.)

**4:**

*Creating Dictionaries.* Given a pair of identically-sized lists, say, `[1, 2, 3, ...]`, and `['abc', 'def', 'ghi', ...]`, process all that list data into a single dictionary that looks like: `{1: 'abc', 2: 'def', 3: 'ghi', ...}`.

**5:**

*`userpw2.pw`.* Following problem deals with the program in Example 7.1 a manager of a database of name-password key-value pairs.

(a) Update the script so that a timestamp is also kept with the password indicating date and time of last login. This interface should prompt for login and password and indicate a successful or failed login as before, but if successful, it should update the last login timestamp. If the login occurs within four hours of the last login, tell the user, "You already logged in at: <last_login_timestamp>."

(b) Add an "administration" menu to include the following two menu options: (1) remove a user and (2) display a list of all users in the system and their passwords

(c) The passwords are currently not encrypted. Add password-encryption if so desired (see the `crypt`, `rotor`, or other cryptographic modules)

(d) Add a GUI interface, i.e., Tkinter, on top of this application.

**6:**

*Lists and Dictionaries.* Create a crude stock portfolio database system: There should be at least four data columns: stock ticker symbol, number of shares, purchase price, and current price (you can add more if you wish). Have the user input values for each column to create a single row. Each row should be created as list. Another all-encompassing list will hold all these rows. Once the data is entered, prompt the user for one column to use as the sort metric. Extract the data values of that column into a dictionary as keys, with their corresponding values being the row that contains that key. Be mindful that the sort metric must have non-coincidental keys or else you will lose a row because dictionaries are not allowed to have more than one value with the same key. You may also choose to have additional calculated output, such as percentage gain/loss, current portfolio values, etc.

**7:**

*Inverting Dictionaries.* Take a dictionary as input and return one as output, but the values are now the keys and vice versa.

**8:**

*Human Resources.* Create a simple name and employee number dictionary application. Have the user enter a list of names and employee numbers. Your interface should allow a sorted output (sorted by name) that displays employee names followed by their employee numbers. EXTRA CREDIT: come up with an additional feature that allows for output to be sorted by employee numbers.

**9:**

*Translations.*

(a) Create a character translator (that works similar to the Unix `tr` command). This function, which we will call `tr()`, takes three strings as arguments: source, destination, and base strings, and has the following declaration:

```
def tr(srcstr, dststr, string)
```

`srcstr` contains the set of characters you want "translated," `dststr` contains the set of characters to translate to, and `string` is the string to perform the translation on. For example, if `srcstr == 'abc'`, `dststr == 'mno'`, and `string == 'abcdef'`, then `tr()` would output `'mno-def'`. Note that `len(srcstr) == len(dststr)`. For this exercise, you can use the `chr()` and `ord()` built-in functions, but they are not necessary to arrive at a solution.

(b) Add a new flag argument to this function to perform case-insensitive

translations.

(c) Update your solution so that it can process character deletions. Any extra characters in `srcstr` which are beyond those which could be mapped to characters in `dststr` should be filtered. In other words, these characters are mapped to no characters in `dststr`, and are thus filtered from the modified string which is returned. For example, if `srcstr == 'abcdef'`, `dststr == 'mno'`, and `string == 'abcdefghi'`, then `tr()` would output `'mnoghi'`. Note now that `len(srcstr) >= len(dststr)`.

**10:**

*Encryption.* Using your solution to the previous problem, and create a "rot13" translator. "rot13" is an old and fairly simplistic encryption routine where by each letter of the alphabet is rotated 13 characters. Letters in the first half of the alphabet will be rotated to the equivalent letter in the second half and vice versa, retaining case. For example, 'a' goes to 'n' and 'X' goes to 'K'. Obviously, numbers and symbols are immune from translation.

(b) Add an application on top of your solution to prompt the user for strings to encrypt (and decrypt on reapplication of the algorithm), as in the following examples:

```
% rot13.py
Enter string to rot13: This is a short sentence.
Your string to en/decrypt was: [This is a short
sentence.].
The rot13 string is: [Guvf vf n fubeg fragrapr.].
%
% rot13.py
Enter string to rot13: Guvf vf n fubeg fragrapr.
Your string to en/decrypt was: [Guvf vf n fubeg
fragrapr.].
The rot13 string is: [This is a short sentence.].
```

# Chapter 8. Conditionals and Loops

The primary focus of this chapter are Python's conditional and looping statements, and all their related components. We will take a close look at **if, while, for,** and their friends **else, elif, break, continue,** and **pass.**

## if statement

The **if** statement for Python will seem amazingly familiar; it is made up of three main components: the keyword itself, an expression which is tested for its truth value, and a code suite to execute if the expression evaluates to non-zero or true. The syntax for an **if** statement:

```
if expression:
    expr_true_suite
```

The suite of the **if** clause, *expr_true_suite,* will be executed only if the above conditional expression results in a Boolean true value. Otherwise, execution resumes at the next statement following the suite.

### Multiple Conditional Expressions

The Boolean operators **and, or,** and **not** can be used to provide multiple conditional expressions or perform negation of expressions in the same **if** statement.

```
if not warn and (system_load >= 10):
    print "WARNING: losing resources"
    warn = warn + 1
```

### Single Statement Suites

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement:

```
if (make_hard_copy == 1): send_data_to_printer()
```

Single line **if** statements such as the above are valid syntax-wise; however, although it may be convenient, it may make your code more difficult to read, so I recommend you indent the suite on the next line. Another good reason is that if you must add another line to the suite, you have to move that line down to the next anyway.

## **else** Statement

Like other languages, Python features an **else** statement that can be paired with an **if** statement. The **else** statement identifies a block of code to be executed if the conditional expression of the **if** statement resolves to a false Boolean value. The syntax is what you expect:

```
if expression:
    expr_true_suite
else:
    expr_false_suite
```

Now the obligatory usage example:

```
if passwd == user.passwd:
    ret_str = "password accepted"
    id = user.id
    valid = 1
else:
    ret_str = "invalid password entered… try again!"
    valid = 0
```

## Dangling **else** Avoidance

Python's design of using indentation rather than braces for code block delimitation not only helps to enforce code correctness, but it even aids implicitly in avoiding potential problems in code that *is* syntactically correct. One of those such problems is the (in)famous "dangling else" problem, a semantic optical illusion.

We present some C code here to illustrate our example (which is also illuminated by K&R and other programming texts):

```
/* dangling-else in C */
if (balance > 0.00)
    if (((balance - amt) > min_bal) && (atm_cashout() == 1))
      printf("Here's your cash; please take all bills.\n");
```

```
else
    printf("Your balance is zero or negative.\n");
```

The question is, which **if** does the else belong to? In the C language, the rule is that the **else** stays with the closest **if.** In our example above, although indented for the outer **if** statement, the **else** statement really belongs to the inner **if** statement because the C compiler ignores superfluous whitespace. As a result, if you have a positive balance but is below the minimum, you will get the horrid (and erroneous) message that your balance is either zero or negative.

Although solving this problem may be facile due to the simplistic nature of the example, any larger sections of code embedded within this framework may be a hair-pulling experience to root out. Python puts up guardrails to not necessarily prevent you from driving off the cliff, but to steer you away from danger. The same example in Python will result in one of the following choices (one of which is correct):

```
if (balance > 0.00):
    if ((balance - amt) > min_bal) and (atm_cashout() == 1):
        print "here's your cash; please take all bills."
else:
    print "your balance is zero or negative"
```

or

```
if (balance > 0.00):
    if ((balance - amt) > min_bal) and (atm_cashout() == 1):
        print "here's your cash; please take all bills."
    else:
        print "your balance is zero or negative"
```

Python's use of indentation forces the proper alignment of code, giving the programmer the ability to make a conscious decision as to which **if** an **else** statement belongs to. By limiting your choices and thus reducing ambiguities, Python encourages you to develop correct code the first time. It is impossible to create a dangling-else problem in Python.

## **elif** (a.k.a. **else-if**) Statement

**elif** is the Python else-if statement. It allows one to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to true. Like the **else,** the **elif** statement is optional. However, unlike **else,** for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if.**

```
if                                                  expression1:
   expr1_true_suite
elif                                                expression2:
   expr2_true_suite
        :
elif                                                expressionN:
   exprN_true_suite
else:
   none_of_the_above_suite
```

At this time, Python does not currently support `switch` or `case` statements as in other languages. Python syntax does not present roadblocks to readability in the presence of a good number of **`if-elif`** statements.

```
if (user.cmd == 'create'):
    action = "create item"
    valid = 1

elif (user.cmd == 'delete'):
    action = 'delete item'
    valid = 1

elif (user.cmd == 'quit'):
    action = 'quit item'
    valid = 1

else:
    action = "invalid choice… try again!"
    valid = 0
```

Python presents an elegant alternative to the `switch/case` statement in the **`for`** statement. Using **`for`**, one can "simulate" switches by cycling through each potential "case," and take action when warranted. (See Section 8.5.3.)

## **`while`** Statement

Python's **`while`** is the first looping statement we will look at in this chapter. In fact, it is a conditional looping statement. In comparison with an **`if`** statement where a true expression will result in a single execution of the **`if`** clause suite, the suite in a **`while`** clause will be executed continuously in a loop until that condition is no longer satisfied.

### General Syntax

Here is the general syntax for a **while** loop:

```
while
                                                    expression:
    suite_to_repeat
```

The *suite_to_repeat* clause of the **while** loop will be executed continuously in a loop until expression evaluates to Boolean false. This type of looping mechanism is often used in a counting situation, such as the example in the next subsection.

## Counting Loops

```
count = 0
while (count < 9):
    print 'the index is:', count
    count = count + 1
```

The suite here, consisting of the **print** and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then bumped up by 1. If we take this snippet of code to the Python interpreter, entering the source and seeing the resulting execution would look something like:

```
>>> count = 0
>>> while (count < 9):
…         print 'the index is:', count
…         count = count + 1
…
the index is: 0
the index is: 1
the index is: 2
the index is: 3
the index is: 4
the index is: 5
the index is: 6
the index is: 7
the index is: 8
```

## Infinite Loops

One must use caution when using **while** loops because of the possibility that this condition never resolves to a false value. In such cases, we would have a loop that never ends on our hands. These "infinite" loops are not necessarily bad things… many

communications "servers" that are part of client-server systems work exactly in that fashion. It all depends on whether or not the loop was meant to run forever, and if not, whether the loop has the possibility of terminating; in other words, will the expression ever be able to evaluate to false?

```
while 1:
    handle, indata = wait_for_client_connect()
    outdata = process_request(indata)
    ack_result_to_client(handle, outdata)
```

For example, the piece of code above was set deliberately to never end because the value 1 will never evaluate to Boolean false. The main point of this server code is to sit and wait for clients to connect, presumably over a network link. These clients send requests which the server understands and processes. After the request has been serviced, a return value or data is returned to the client who may either drop the connection altogether or send another request. As far as the server is concerned, it has performed its duty to this one client and returns to the top of the loop to wait for the next client to come along. You will find out more about client-server computing in the Networking and Web Programming chapters 16 and 19).

### Single Statement Suites

Similar to the `if` statement syntax, if your `while` clause consists only of a single statement, it may be placed on the same line as the `while` header. Here is an example of a one-line `while` clause:

```
while not ready: ready = is_data_ready()
```

## `for` Statement

The other looping mechanism in Python comes to us in the form of the `for` statement. Unlike the traditional conditional looping `for` statement found in mainstream third-generation languages (3GLs) like C, Fortran, or Pascal, Python's `for` is more akin to a scripting language's iterative `foreach` loop.

### General Syntax

Iterative loops index through individual elements of a set and terminate when all the items are exhausted. Python's `for` statement iterates only through sequences, as indicated in the general syntax here:

```
for iter_var in sequence:
```

```
    suite_to_repeat
```

The sequence *sequence* will be iterated over, and with each loop, the *iter_var* iteration variable is set to the current element of the sequence, presumably for use in *suite_to_repeat*.

## Used with Sequence Types

In this section, we will see how the **for** loop works with the different sequence types. The examples will include string, list, and tuple types.

```
>>> for eachLetter in 'Names':
…       print 'current letter:', eachLetter
…
current letter: N
current letter: a
current letter: m
current letter: e
current letter: s
```

When iterating over a string, the iteration variable will always consist of only single characters (strings of length 1). Such constructs may not necessarily be useful. When seeking characters in a string, more often than not, the programmer will either use **in** to test for membership, or one of the string module functions or string methods to check for substrings.

One place where seeing individual characters does come in handy is during the debugging of sequences in a **for** loop in an application where you are expecting strings or entire objects to show up in your **print** statements. If you see individual characters, this is usually a sign that you received a single string rather than a sequence of objects.

There are two basic ways of iterating over a sequence:

### Iterating by Sequence Item

```
>>> nameList ['Walter', "Nicole", 'Steven', 'Henry']
>>> for eachName in nameList:
…        print eachName, "Lim"
…
Walter Lim
Nicole Lim
Steven Lim
Henry Lim
```

In the above example, a list is iterated over, and for each iteration, the `eachName` variable contains the list element that we are on for that particular iteration of the loop.

### Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself:

```
>>> nameList = ['Shirley', "Terry", 'Joe', 'Heather', 'Lucy']
>>> for nameIndex in range(len(nameList)):
…       print "Liu,", nameList[nameIndex]
…
Liu, Shirley
Liu, Terry
Liu, Joe
Liu, Heather
Liu, Lucy
```

Rather than iterating through the elements themselves, we are iterating through the indices of the list.

We employ the assistance of the **`len()`** built-in function, which provides the total number of elements in the tuple as well as the `range()` built-in function (which we will discuss in more detail below) to give us the actual sequence to iterate over.

```
>>> len(nameList)
5
>>> range(len(nameList))
[0, 1, 2, 3, 4]
```

Using `range()`, we obtain a list of the indexes that `nameIndex` iterates over; and using the slice/subscript operator ( `[ ]` ), we can obtain the corresponding sequence element.

Those of you who are performance pundits will no doubt recognize that iteration by sequence item wins over iterating via index. If not, this is something to think about. (See Exercise 8-13).

## **`Switch/Case`** Statement Proxy

Earlier in Section 8.3, we introduced the **`if-elif-else`** construct and indicated that Python did not support a `switch/case` statement. In many cases, an incredibly long set of **`if-elif-else`** statements can be replaced by a **`for`** loop, which contains the "case" items in a sequence which is iterated over. We present a modified version of the example in Section 8.3, moving all the **`elif`** statements into the **`for`** loop:

```
for cmd in ('add', 'delete', 'quit'):
    if cmd == user.cmd:
        action = cmd + " item"
        valid = 1
        break
else:
    action = "invalid choice… try again!"
    valid = 0
```

You are now probably glad to see that there is some kind of substitute for the lack of a `switch/case` statement in Python, but do you realize that using a list gives you even more power as a programmer? In other languages, the elements of a `case` statement are constant and a static part of the code. By using lists in Python, not only can these elements be variables, but they can also be dynamic and changed during run-time!

Final note, it may have surprised you to see an **`else`** statement at the end there.Yes, **`else`** statements can be used with **`for`** loops. In this case, the **`else`** clause is executed only if the **`for`** loop finished to completion. More on **`else`** coming up in Section 8.9.

## `range()` [and `xrange()`] Built-in Function(s)

We mentioned above during our introduction to Python's **`for`** loop that it is an iterative looping mechanism. Python also provides a tool that will let us use the **`for`** statement in a traditional pseudo-conditional setting, i.e., when counting from one number to another and quitting once the final number has been reached or some condition is no longer satisfied.

The built-in function `range()` can turn your `foreach`-like **`for`**-loop back into one that you are more familiar with, i.e., counting from zero to ten, or counting from 10 to 100 in increments of 5.

### `range()` Full Syntax

Python presents two different ways to use `range()`. The full syntax requires that two or all three integer arguments are present:

```
range( start, end, step=1)
```

`range()` will then return a list where for any `k, start <= k < end` and `k` iterates from `start` to `end` in increments of `step. step` cannot be 0, or else an error condition will occur.

```
>>> range(2, 19, 3)
```

```
[2, 5, 8, 11, 14, 17]
```

If step is omitted and only two arguments given, `step` takes a default value of 1.

```
>>> range(3,7)
[3, 4, 5, 6]
```

Let's take a look at an example used in the interpreter environment:

```
>>> for eachVal in range(2, 19, 3):
…    print "value is:", eachVal
…
value is: 2
value is: 5
value is: 8
value is: 11
value is: 14
value is: 17
```

Our **`for`** loop now "counts" from two to nineteen, incrementing by steps of three. If you are familiar with C, then you will notice the direct correlation between the arguments of `range()` and those of the variables in the C **`for`** loop:

```
/* equivalent loop in C */
for (eachVal = 2; eachVal < 19; i += 3) {
   printf("value is: %d\n", eachVal);
}
```

Although it seems like a conditional loop now (checking if `eachVal<19`), reality tells us that `range()` takes our conditions and generates a list that meets our criteria, which in turn, is used by the same Python **`for`** statement.

### `range()` Abbreviated Syntax

`range()` also has a simple format, which takes one or both integer arguments:

```
range( start=0, end)
```

Given both values, this shortened version of `range()` is exactly the same as the long version of `range()` taking two parameters with `step` defaulting to 1. However, if given only a single value, `start` defaults to zero, and `range()` returns a list of numbers from zero up to the argument `end`:

```
>>> range(5)
[0, 1, 2, 3, 4]
```

We will now take this to the Python interpreter and plug in **`for`** and **`print`** statements to arrive at:

```
>>> for count in range(5):
…       print count
…
0
1
2
3
4
```

Once **`range()`** executes and produces its list result, our expression above is equivalent to the following:

```
>>> for count in [0, 1, 2, 3, 4]:
…     print count
```

**NOTE**

Now that you know both syntaxes for **`range()`**, one nagging question you may have is, why not just combine the two into a single one that looks like this?

```
range( start=0, end, step=1)# invalid
```

This syntax will work for a single argument or all three, but not two. It is illegal because the presence of **`step`** requires `start` to be given. In other words, you cannot provide **`end`** and **`step`** in a two-argument version because they will be (mis)interpreted as **`start`** and **`end`**.

### `xrange()` Function for Limited Memory Situations

`xrange()` is similar to `range()` except that if you have a really large range list, `xrange()` may come in more handy because it does not have to make a complete copy of the list in memory. This built-in was made for exclusive use in `for` loops. It doesn't make sense outside a `for` loop. Also, as you can imagine, the performance will not be as good because the entire list is *not* in memory.

Now that we've covered all the loops Python has to offer, let us take a look at the peripheral commands that typically go together with loops. These include statements to abandon the loop (`break`) and to immediately begin the next iteration (`continue`).

## `break` Statement

The `break` statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional `break` found in C. The most common use for `break` is when some external condition is triggered (usually by testing with an `if` statement), requiring a hasty exit from a loop. The `break` statement can be used in both `while` and `for` loops.

```
count = num / 2
while count > 0:
    if (num % count == 0):
        print count, 'is the largest factor of', num
        break
    count = count - 1
```

The task of this piece of code is to find the largest divisor of a given number `num`. We iterate through all possible numbers that could possibly be factors of `num`, using the `count` variable and decrementing for every value that does NOT divide `num`. The first number that evenly divides `num` is the largest factor, and once that number is found, we no longer need to continue and use `break` to terminate the loop.

```
phone2remove = '555-1212'
for eachPhone in phoneList:
    if eachPhone == phone2remove:
        print "found", phone2remove, '… deleting'
        deleteFromPhoneDB(phone2remove)
        break
```

The `break` statement here is used to interrupt the iteration of the list. The goal is to find a target element in the list, and, if found, to remove it from the database and break out of the loop.

## `continue` Statement

**NOTE**

Whether in Python, C, Java, or any other structured language which features the `continue` statement, there is a misconception among some beginning programmers that the traditional `continue` statement "immediately starts the next iteration of a loop." While this may seem to be the apparent action, we would like to clarify this somewhat invalid supposition. Rather than beginning the next iteration of the loop when a `continue` statement is encountered, a `continue` statement terminates or discards the remaining statements in the current loop iteration and goes back to the top.

If we are in a conditional loop, the conditional expression is checked for validity before beginning the next iteration of the loop. Once confirmed, then the next iteration begins. Likewise, if the loop were iterative, a determination must be made as to whether there are any more arguments to iterate over. Only when that validation has completed successfully can we begin the next iteration.

The `continue` statement in Python is not unlike the traditional `continue` found in other high-level languages. The `continue` statement can be used in both `while` and `for` loops. The `while` loop is conditional, and the `for` loop is iterative, so using `continue` is subject to the same requirements (as highlighted in the Core Note above) before the next iteration of the loop can begin. Otherwise, the loop will terminate normally.

```
valid = 0
count = 3
while count > 0:
    input = raw_input("enter password")
    # check for valid passwd
    for eachPasswd in passwdList:
        if input == eachPasswd:
            valid = 1
            break
    if not valid:    # (or valid == 0)
        print "invalid input"
        count = count - 1
        continue
    else:
        break
```

In this combined example using `while`, `for`, `if`, `break`, and `continue`, we are looking at validating user input. The user is given three opportunities to enter the correct

password; otherwise, the `valid` variable remains a false value of 0, which presumably will result in appropriate action being taken soon after.

## `pass` Statement

One Python statement not found in C is the `pass` statement. Because Python does not use curly braces to delimit blocks of code, there are places where code is syntactically required. We do not have the equivalent empty braces or single semicolon the way C has to indicate "do nothing." If you use a Python statement that expects a sub-block of code or suite, and one is not present, you will get a syntax error condition. For this reason, we have `pass`, a statement that does absolutely nothing—it is a true NOP, to steal the "No OPeration" assembly code jargon. Style- and development-wise, `pass` is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

```
def foo_func():
    pass
```

or

```
if user_choice == 'do_calc':
    pass
else:
    pass
```

This code structure is helpful during the development or debugging stages because you want the structure to be there while the code is being created, but you do not want it to interfere with the other parts of the code that have been completed already. In places where you want nothing to execute, `pass` is a good tool to have in the box.

Another popular place is with exception handling, which we will take a look at in Chapter 10; this is where you can track an error if it occurs, but take no action if it is not fatal (you just want to keep a record of the event or perform an operation under the covers if an error occurs).

## `else` Statement... Take Two

In C (as well as in most other languages), you will *not* find an `else` statement outside the realm of conditional statements, yet Python bucks the trend again by offering these in

**while** or **for** loops. How do they work? When used with loops, an **else** clause will be executed only if a loop finishes to completion, meaning they were not abandoned by **break.**

One popular example of **else** usage in a **while** statement is in finding the largest factor of a number. We have implemented a function which performs this task, using the **else** statement with our **while** loop. The `showMaxFactor()` function in Example 8.1 (`maxFact.py`) utilizes the **else** statement as part of a **while** loop.

**Example 8.1. while-else Loop Example (maxFact.py)**

This program displays the largest factors for numbers between 10 and 20. If the number is prime, the script will indicate that as well.

```
 <$nopage>
001 1    #!/usr/bin/env python
002 2
003 3    def showMaxFactor(num):
004 4        count = num / 2
005 5        while count > 1:
006 6            if (num % count == 0):
007 7                print 'largest factor of %d is %d' % \
008 8                    (num, count)
009 9                break <$nopage>
010 10           count = count - 1
011 11       else: <$nopage>
012 12           print num, "is prime"
013 13
014 14   for eachNum in range(10, 21):
015 15       showMaxFactor(eachNum)
016  <$nopage>
```

The loop beginning on line 3 in `showMaxFactor()` counts down from half the amount (starts checking if two divides the number, which would give the largest factor). The loop decrements each time (line 10) through until a divisor is found (lines 6–9). If a divisor has not been found by the time the loop decrements to 1, then the original number must be prime. The **else** clause on lines 11–12 takes care of this case. The main part of the program on lines 14–15 fires off the requests to `showMaxFactor()` with the numeric argument.

Running our program results in the following output:

```
largest factor of 10 is 5
11 is prime
largest factor of 12 is 6
13 is prime
largest factor of 14 is 7
largest factor of 15 is 5
largest factor of 16 is 8
17 is prime
largest factor of 18 is 9
```

```
19 is prime
largest factor of 20 is 10
```

Likewise, a **for** loop can have a post-processing **else.** It operates exactly the same way as for a **while** loop. As long as the **for** loop exits normally (not via **break**), the **else** clause will be executed. We saw such an example in Section 8.5.3.

Table 8.1 summarizes which conditional or looping statements auxiliary statements can be used.

**Table 8.1. Auxiliary Statements to Loops and Conditionals**

| | *Loops and Conditionals* | | |
|---|---|---|---|
| ***Auxiliary Statements*** | **if** | **while** | **for** |
| **elif** | • | | |
| **else** | • | • | • |
| **break** | | • | • |
| **continue** | | • | • |
| **pass**[a] | • | • | • |

[a] `pass` is valid anywhere a suite is required (also includes `elif, else, class, def, try, except, finally`)

## Exercises

**1:**

*Conditionals.* Study the following code:

```
# statement A
if x > 0:
     # statement B
     pass

elif x < 0:
     # statement C
     pass

else:
    # statement D
    pass

# statement E
```

(a) Which of the statements above (A, B, C, D, E) will be executed if x < 0?

(b) Which of the statements above will be executed if x == 0?

(c) Which of the statements above will be executed if x > 0?

**2:**

*Loops.* Write a program to have the user input three (3) numbers: (**f**)rom, (**t**)o, and (**i**)ncrement. Count from **f** to **t** in increments of **i**, inclusive of **f** and **t**. For example, if the input is **f** == 2, **t** == 24, and **i** == 4, the program would output: 2, 6, 10, 14, 18, 22.

**3:**

*`range()`.* What argument(s) could we give to the `range()` built-in function if we wanted the following lists to be generated?

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

(b) `[3, 6, 9, 12, 15, 18]`

(c) `[-20, 200, 420, 640, 860]`

**4:**

*Prime Numbers.* We presented some code in this chapter to determine a number's largest factor or if it is prime. Turn this code into a Boolean function called `isprime()` such that the input is a single value, and the result returned is 1 if the number is prime and 0 otherwise.

**5:**

*Factors.* Write a function called `getfactors()` that takes a single integer as an argument and returns a list of all its factors, including 1 and itself.

**6:**

*PrimeFactorization.* Take your solutions for `isprime()` and `getfactors()` in the previous problems and create a function that takes an integer as input and returns a list of its prime factors. This process, known as prime factorization, should output a list of factors such that if multiplied together, they will result in the original number. Note that there could be repeats in the list. So if you gave an input of 20, the output would be [2, 2, 5].

**7:**

*Perfect Numbers.* A perfect number is one whose factors (except itself) sum to itself. For example, the factors of 6 are 1, 2, 3, and 6. Since 1 + 2 + 3 is 6, it (6) is considered a perfect number. Write a function called `isperfect()` which takes a

single integer input and outputs 1 if the number is perfect and 0 otherwise.

**8:**

*Factorial.* The factorial of a number is defined as the product of all values from one to that number. A shorthand for N factorial is N! where N! == factorial (N) == 1 * 2 * 3 * … * (N-2) * (N-1) * N!. So 4! == 1 * 2 * 3 * 4. Write a routine such that given N, the value N! is returned.

**9:**

*Fibonacci Numbers.* The Fibonacci number sequence is 1, 1, 2, 3, 5, 8, 13, 21, etc. In other words, the next value of the sequence is the sum of the previous two values in the sequence.

**10:**

*Text Processing.* Determine the total number of vowels, consonants, and words (separated by spaces) in a text sentence. Ignore special cases for vowels and consonants such as "h," "y," "qu," etc.

**11:**

*Text Processing.* Write a program to ask the user to input a list of names, in the format "Last Name, First Name," i.e., last name, comma, first name. Write a function that manages the input so that when/if the user types the names in the wrong order, i.e., "First Name Last Name," the error is corrected, and the user is notified. This function should also keep track of the number of input mistakes. When the user is done, sort the list, and display the sorted names in "Last Name, First Name" order.

EXAMPLE input and output: (you don't have to do it this way exactly)

```
% nametrack.py
Enter total number of names: 5

Please enter name 0: Smith, Joe
Please enter name 1: Mary Wong
>> Wrong format…   should be Last, First
>> You have done this 1 time(s) already.  Fixing input…
Please enter name 2: Hamilton, Gerald
Please enter name 3: Royce, Linda
Please enter name 4: Winston Salem
>> Wrong format…   should be Last, First
>> You have done this 2 time(s) already.  Fixing input…

The sorted list (by last name) is:
    Hamilton, Gerald
    Royce, Linda
    Salem, Winston
    Smith, Joe
```

```
    Wong, Mary
```

**12:**

*(Integer) Bit Operators.* Write a program that takes begin and end values and prints out a decimal, binary, octal, hexadecimal chart like below. If any of the characters are printable ASCII characters, then print those, too. If none is, you may omit the ASCII column header.

```
SAMPLE OUTPUT 1
---------------
Enter begin value: 9
Enter end value: 18
DEC    BIN    OCT    HEX
--------------------------
  9    01001   11     9
 10    01010   12     a
 11    01011   13     b
 12    01100   14     c
 13    01101   15     d
 14    01110   16     e
 15    01111   17     f
 16    10000   20    10
 17    10001   21    11
 18    10010   22    12
SAMPLE OUTPUT 2
---------------
Enter begin value: 26
Enter end value: 41
DEC    BIN     OCT    HEX    ASCII
-------------------------------------
 26   011010    32    1a
 27   011011    33    1b
 28   011100    34    1c
 29   011101    35    1d
 30   011110    36    1e
 31   011111    37    1f
 32   100000    40    20
 33   100001    41    21     !
 34   100010    42    22     "
 35   100011    43    23     #
 36   100100    44    24     $
 37   100101    45    25     %
 38   100110    46    26     &
 39   100111    47    27     '
 40   101000    50    28     (
 41   101001    51    29     )
```

**13:**

*Performance.* In Section 8.5.2, we examined two basic ways of iterating over a sequence: (1) by sequence item, and (2) via sequence index. We pointed out at the

end that the latter does not perform as well over the long haul (on my system here, a test suite shows performance is nearly twice as bad [83% worse]). Why do you think that is, and what are the reasons?

# Chapter 9. Files and Input/Output

This chapter is intended to give you an in-depth introduction on the use of files and related input/output capabilities of Python. We introduce the file object (its built-in function, and built-in methods and attributes), review the standard files, discuss accessing the file system, hint at file execution, and briefly mention persistent storage and modules in the standard library related to "file-mania."

## File Objects

File objects can be used not only to access normal disk files, but also any other type of "file" that uses that abstraction. Once the proper "hooks" are installed, you can access other objects with file-like interfaces in the same manner you would access normal files.

The `open()` built-in function (see below) returns a file object which is then used for all succeeding operations on the file in question. There are a large number of other functions which return a file or file-like object. One primary reason for this abstraction is that many input/output data structures prefer to adhere to a common interface. It provides consistency in behavior as well as implementation. Operating systems like Unix even feature files as an underlying and architectural interface for communication. Remember, files are simply a contiguous sequence of bytes. Anywhere data needs to be sent usually involves a byte stream of some sort, whether the stream occurs as individual bytes or blocks of data.

## File Built-in Function [`open()`]

As the key to opening file doors, the `open()` built-in function provides a general interface to initiate the file input/output (I/O) process. `open()` returns a file object on a successful opening of the file or else results in an error situation. When a failure occurs, Python generates or *raises* an `IOError` exception—we will cover errors and exceptions in the next chapter. The basic syntax of the `open()` built-in function is:

```
file_object = open(file_name, access_mode='r', buffering=-1)
```

The *file_name* is a string containing the name of the file to open. It can be a relative or absolute/full pathname. The *access_mode* optional variable is also a string, consisting of a set of flags indicating which mode to open the file with. Generally, files are opened with the modes "r," "w," or "a," representing read, write, and append, respectively.

Any file opened with mode "r" must exist. Any file opened with "w" will be truncated first if it exists, and then the file is (re)created. Any file opened with "a" will be opened for write. If the file exists, the initial position for file (write) access is set to the end-of-file. If the file does not exist, it will be created, making it the same as if you opened the file in "w" mode. If you are a C programmer, these are the same file open modes used for the C library function `fopen()`.

There are other modes supported by `fopen()` that will work with Python's `open()`. These include the "+" for read-write access and "b" for binary access. One note regarding the binary flag: "b" is antiquated on all Unix systems which are POSIX-compliant (including Linux) because they treat all files as "binary" files, including text files. Here is an entry from the Linux manual page for `fopen()`, which is from where the Python `open()` function is derived:

The mode string can also include the letter "b" either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with ANSI C3.159-1989 ("ANSI C") and has no effect; the "b" is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the "b" may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-Unix environments.)

You will find a complete list of file access modes, including the use of "b" if you choose to use it, in Table9.1. If *access_mode* is not given, it defaults automatically to "r."

The other optional argument, *buffering*, is used to indicate the type of buffering that should be performed when accessing the file. A value of 0 means no buffering should occur, a value of 1 signals line buffering, and any value greater than 1 indicates buffered I/O with the given value as the buffer size. The lack of or a negative value indicates that the system default buffering scheme should be used, which is line buffering for any teletype or tty-like device and normal buffering for everything else. Under normal circumstances, a *buffering* value is not given, thus using the system default.

**Table 9.1. Access Modes for File Objects**

| *File Mode* | *Operation* |
|---|---|
| r | open for read |
| w | open for write (truncate if necessary) |
| a | open for write (start at EOF, create if necessary) |
| r+ | open for read and write |
| w+ | open for read and write (see "w" above) |
| a+ | open for read and write (see "a" above) |
| rb | open for binary read |

| | |
|---|---|
| wb | open for binary write (see "w" above) |
| ab | open for binary append (see "a" above) |
| rb+ | open for binary read and write (see "r+" above) |
| wb+ | open for binary read and write (see "w+" above) |
| ab+ | open for binary read and write (see "a+" above) |

Here are some examples for opening files:

```
fp = open('/etc/motd')         #open file for read
fp = open('test',  'w')        #open file for write
fp = open('data',  'r+')       #open file for read/write
fp = open('c:\io.sys', 'rb')   #open binary file for read
```

## File Built-in Methods

Once `open()` has completed successfully and returned a file object, all subsequent access to the file transpires with that "handle." File methods come in four different categories: input, output, movement within a file, which we will call "intra-file motion," and miscellaneous. A summary of all file methods can be found in Table 9.3. We will now go over each category.

### Input

The `read()` method is used to read bytes directly into a string, reading at most the number of bytes indicated. If no size is given, the default value is set to -1, meaning that the file is read to the end. The `readline()` method reads one line of the open file (reads all bytes until a NEWLINE character is encountered). The NEWLINE character is retained in the returned string. The `readlines()` method is similar, but reads all remaining lines as strings and returns a list containing the read set of lines. The `readinto()` method reads the given number of bytes into a writable buffer object, the same type of object returned by the unsupported `buffer()` built-in function. (Since `buffer()` is not supported, neither is `readinto()` ).

### Output

The `write()` built-in method has the opposite functionality as `read()` and `readline()`. It takes a string which can consist of one or more lines of text data or a block of bytes and writes the data to the file. `writelines()` operates on a list just like `readlines()`, but takes a list of strings and writes them out to a file. NEWLINE characters are not inserted between each line; so if desired, they must be added to the end of each line before `writelines()` is called.

This is easily accomplished in Python 2.0 with a list comprehension:

```
>>> output=['1stline', '2ndline', 'the end']
>>> [x + '\n' for x in output]
['1stline\012', '2ndline\012', 'the end\012']
```

Note that there is no "`writeline()`" method since it would be equivalent to calling `write()` with a single line string terminated with a NEWLINE character.

## Intra-file Motion

The `seek()` method (analogous to the `fseek()` function in C) moves the file pointer to different positions within the file. The offset in bytes is given along with a *relative offset* location called *`whence`*. A value of 0 indicates distance from the beginning of a file (note that a position measured from the beginning of a file is also known as the *absolute offset*), a value of 1 indicates movement from the current location in the file, and a value of 2 indicates that the offset is from the end of the file. If you have used `fseek()` as a C programmer, the values 0, 1, and 2 correspond directly to the constants SEEK_SET, SEEK_CUR, and SEEK_END, respectively. Use of the `seek()` method comes to play when opening a file for read and write access.

`tell()` is a complementary method to `seek();` it tells you the current location of the file—in bytes from the beginning of the file.

## Others

The `close()` method completes access to a file by closing it. The Python garbage collection routine will also close a file when the file object reference has decreased to zero. One way this can happen is when only one reference exists to a file, say, `fp = open(),` and `fp` is reassigned to another file object before the original file is explicitly closed. Good programming style suggests closing the file before reassignment to another file object.

The `fileno()` method passes back the file descriptor to the open file. This is an integer argument that can be used in lower-level operations such as those featured in the `os` module. The `flush()` method. `isatty()` is a Boolean built-in method that returns 1 if the file is a tty-like device and 0 otherwise. The `truncate()` method truncates the file to 0 or the given size bytes.

## File Method Miscellany

We will now reprise our first file example from Chapter 2:

```
filename = raw_input('Enter file name: ')
file = open(filename,  'r')
allLines = file.readlines()
file.close()
```

```
for eachLine in allLines:
    print eachline,
```

We originally described how this program differs from most standard file access in that all the lines are read ahead of time before any display to the screen occurs. Obviously, this is not advantageous if the file is large. In those cases, it may be a good idea to go back to the tried-and-true way of reading and displaying one line at a time:

```
filename = raw_input('Enter file name: ')
file = open(filename,  'r')
done = 0
while not done:
    aLine = file.readline()
    if aLine != " ":
        print aLine,
    else:
        done = 1
file.close()
```

In this example, we do not know when we will reach the end of the file, so we create a Boolean flag `done`, which is initially set for false. When we reach the end of the file, we will reset this value to true so that the `while` loop will exit. We change from using `readlines()` to read all lines to `readline()`, which reads only a single line. `readline()` will return a blank line if the end of the file has been reached. Otherwise, the line is displayed to the screen.

We anticipate a burning question you may have… "Wait a minute! What if I have a blank line in my file? Will Python stop and think it has reached the end of my file?" The answer is, of course, no. A blank line in your file will not come back as a blank line. Recall that every line has one or more line separator characters at the end of the line, so a "blank line" would consist of a NEWLINE character or whatever your system uses. So even if the line in your text file is "blank," the line which is read is *not* blank, meaning your application would not terminate until it reaches the end-of-file.

**NOTE**

One of the inconsistencies of operating systems is the line separator character which their file systems support. On Unix, the line separator is the NEWLINE ( `\n`) character. For the Macintosh, it is the RETURN (`\r`), and DOS and Windows uses both ( `\r\n`). Check your operating system to determine what your line separator(s) are.

Other differences include the file pathname separator (Unix uses '`/`', DOS and Windows use '`\`', and the Macintosh uses '`:`'), the separator used to delimit a set of file pathnames, and the denotations for the current and parent directories.

These inconsistencies generally add an irritating level of annoyance when creating applications that run on all three platforms (and more if more architectures and operating systems are supported). Fortunately, the designers of the `os` module in Python have thought of this for us. The `os` module has five attributes which you may find useful. They are listed below in *Table 9.2*.

**Table 9.2. `os` Module Attributes to Aid in Multi-platform Development**

| ***`os` Module Attribute*** | ***Description*** |
|---|---|
| `linesep` | string used to separate lines in a file |
| `sep` | string used to separate file pathname components |
| `pathsep` | string used to delimit a set of file pathnames |
| `curdir` | string name for current working directory |
| `pardir` | string name for parent (of current working directory) |

Regardless of your platform, these variables will be set to the correct values when you import the `os` module. One less headache to worry about.

We would also like to remind you that the comma placed at the end of the **`print`** statement is to suppress the NEWLINE character that **`print`** normally adds at the end of output. The reason for this is because every line from the text file already contains a NEWLINE. `readline()` and `readlines()` do not strip off any whitespace characters in your line (see exercises.) If we omitted the comma, then your text file display would be doublespaced one NEWLINE which is part of the input and another added by the **`print`** statement.

Before moving on to the next section, we will show two more examples, the first highlighting output to files (rather than input), and the second performing both file input and output as well as using the `seek()` and `tell()` methods for file positioning.

```
filename = raw_input('Enter file name: ')
file = open(filename,  'w')
done = 0
while not done:
  aLine = raw_input("Enter a line ('.' to quit): ")
  if aLine != ".":
      file.write(aLine + '\n')
  else:
      done = 1
file.close()
```

This piece of code is practically the opposite of the previous. Rather than reading one line at a time and displaying it, we ask the user for one line at a time, and send them out to the

file. Our call to the `write()` method must contain a NEWLINE because `raw_input()` does not preserve it from the user input. Because it may not be easy to generate an end-of-file character from the keyboard, the program uses the period ( . ) as its end-of-file character, which, when entered by the user, will terminate input and close the file.

Our final example opens a file for read and write, creating the file scratch (after perhaps truncating an already-existing file). After writing data to the file, we move around within the file using `seek()`. We also use the `tell()` method to show our movement.

```
>>> f = open('/tmp/x', 'w+')
>>> f.tell()
0
>>> f.write('test line 1\n')     # add 12-char string [0-11]
>>> f.tell()
12
>>> f.write('test line 2\n')     # add 12-char string [12-23]
>>> f.tell()                     # tell us current file location (end))
24
>>> f.seek(-12, 1)               # move back 12 bytes
>>> f.tell()                     # to beginning of line 2
12
>>> f.readline()
'test line 2\012'
>>> f.seek(0, 0)                 # move back to beginning
>>> f.readline()
'test line 1\012'
>>> f.tell()                     # back to line 2 again
12
>>> f.readline()
'test line 2\012'
>>> f.tell()                     # at the end again
24
>>> f.close()                    # close file
```

Table9.3 lists all the built-in methods for file objects:

**Table 9.3. Methods for File Objects**

| *File Object Method* | *Operation* |
|---|---|
| `file.close()` | close `file` |
| `file.fileno()` | return integer file descriptor (FD) for `file` |
| `file.flush()` | flush internal buffer for `file` |
| `file.isatty()` | return 1 if `file` is a tty-like device, 0 otherwise |
| `file.read (size=-1)` | read all or `size` bytes of file as a string and return it |
| `file.readinto(buf, size)` [a] | read *size* bytes from `file` into buffer `buf` |
| `file.readline()` | read and return one line from `file` (includes trailing "\n") |
| `file.readlines()` | read and returns all lines from `file` as a list (includes all |

| | |
|---|---|
| | trailing "\n" characters) |
| *file*.seek(*off*, *whence*) | move to a location within *file*, *off* bytes offset from *whence* (0 == beginning of file, 1 == current location, or 2 == end of file) |
| *file*.tell() | return current location within *file* |
| *file*.truncate(*size*=0) | truncate *file* to 0 or size bytes |
| *file*.write(*str*) | write string *str* to *file* |
| *file*.writelines(*list*) | write *list* of strings to *file* |

[a] unsupported method introduced in Python 1.5.2 (other implementations of file-like objects do not include this method)

## File Built-in Attributes

File objects also have data attributes in addition to its methods. These attributes hold auxiliary data related to the file object they belong to, such as the file naFme (*file*.name), the mode with which the file was opened (*file*.mode), whether the file is closed (*file*.closed), and a flag indicating whether an additional space character needs to be displayed before successive data items when using the **print** statement (*file*.softspace).Table 9.4 lists these attributes along with a brief description of each.

**Table 9.4. Attributes for File Objects**

| *File Object Attribute* | *Description* |
|---|---|
| *file*.closed | 1 if *file* is closed, 0 otherwise |
| *file*.mode | access mode with which *file* was opened |
| *file*.name | name of *file* |
| *file*.softspace | 0 if space explicitly required with print, 1 otherwise; rarely used by the programmer—generally for internal use only |

## Standard Files

There are generally three standard files which are made available to you when your program starts. These are standard input (usually the keyboard), standard output (buffered output to the monitor or display), and standard error (unbuffered output to the screen). (The "buffered" or "unbuffered" output refers to that third argument to open()). These files are named stdin, stdout, and stderr and take after their names from the C language. When we say these files are "available to you when your program starts," that means that these files are pre-opened for you, and access to these files may commence once you have their file handles.

Python makes these file handles available to you from the sys module. Once you import sys, you have access to these files as sys.stdin, sys.stdout, and sys.stderr. The **print** statement normally outputs to sys.stdout while the raw_input() built-in function receives its input from sys.stdin.

We will now take yet another look at the "Hello World!" program so that you can compare the similarities and differences between using **print**/raw_input() and directly with the file names:

### print

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

### sys.stdout.write()

```
import sys
sys.stdout.write('Hello World!' + '\n)
```

Notice that we have to explicitly provide the NEWLINE character to sys.stdout's write() method. In the input examples below, we do not because readline() executed on sys.stdin preserves the readline. raw_input() does not, hence we will allow print to add its NEWLINE.

### raw_input()

```
aString = raw_input('Enter a string: ')
print aString
```

### sys.stdin.readline()

```
import sys
sys.stdout.write('Enter a string: ')
aString = sys.stdin.readline()
sys.stdout.write(aString)
```

## Command-line Arguments

The sys module also provides access to any *command-line arguments* via the sys.argv. Command-line arguments are those arguments given to the program in addition to the script name on invocation. Historically, of course, these arguments are so named because they are given on the command-line along with the program name in a text-based environment like a Unix- or DOS-shell. However, in an IDE or GUI environment, this would not be the case. Most IDEs provide a separate window with which to enter your

"command-line arguments." These, in turn, will be passed into the program as if you started your application from the command-line.

Those of you familiar with C programming may ask, "Where is `argc?`" The strings "argv" and "argv" stand for "argument count" and "argument vector," respectively. The `argv` variable contains an array of strings consisting of each argument from the command-line while the `argc` variable contains the number of arguments entered. In Python, the value for `argc` is simply the number of items in the `sys.argv` list, and the first element of the list, `sys.argv[0],` is always the program name. Summary:

`sys.argv` is the list of command-line arguments

`len(sys.argv)` *is the number of command-line arguments (a.k.a.* `argc`*)*

Let us create a small test program called `argv.py` with the following lines:

```
import sys

print 'you entered', len(sys.argv), 'arguments…'
print 'they were:', str(sys.argv)
```

Here is an example invocation and output of this script:

```
% argv.py 76 tales 85 hawk
you entered 5 arguments…
they were: ['argv.py', '76', 'tales', '85', 'hawk']
```

Are command-line arguments useful? Unix commands are typically programs which take input, perform some function, and send output as a stream of data. This data is usually sent as input directly to the next program, which does some other type of function or calculation and sends the new output to another program, and so on. Rather than saving the output of each program and potentially taking up a good amount of disk space, the output is usually "piped" in to the next program as *its* input. This is accomplished by providing data on the command-line or through standard input. When a program displays or sends output to the standard output file, the result would be displayed on the screen—unless that program is also "piped" to another program, in which case that standard output file is really the standard input file of the next program. I assume you get the drift by now!

Command-line arguments allow a programmer or administrator to start a program perhaps with different behavioral characteristics. Much of the time, this execution takes place in the middle of the night and run as a batch job without human interaction. Command-line arguments and program options enable this type of functionality. As long

as there are computers sitting idle at night and plenty of work to be done, there will always be a need to run programs in the background on our very expensive "calculators."

Python features a `getopt` module that helps you parse command-line options and arguments.

## File System

Access to your file system occurs mostly through the Python `os` module. This module serves as the primary interface to your operating system facilities and services from Python. The `os` module is actually a front-end to the real module that is loaded, a module that is clearly operating system-dependent. This "real" module may be one of the following: `posix` (Unix), `nt` (Windows), `mac` (Macintosh), `dos` (DOS), `os2` (OS/2), etc. You should never import those modules directly. Just import `os` and the appropriate module will be loaded, keeping all the underlying work hidden from sight. Depending on what your system supports, you may not have access to some of the attributes which may be available in other operating system modules.

In addition to managing processes and the process execution environment, the `os` module performs most of the major file system operations that the application developer may wish to take advantage of. These features include removing and renaming files, traversing the directory tree, and managing file accessibility. Table 9.5 lists some of the more common file or directory operations available to you from the `os` module.

A second module that performs specific pathname operations is also available. The `os.path` module is accessible through the `os` module. Included with this module are functions to manage and manipulate file pathname components, obtain file or directory information, and make file path inquiries. Table 9.6 outlines some of the more common functions in `os.path`.

These two modules allow for consistent access to the file system regardless of platform or operating system. The program in Example 9.1 (`ospathex.py`) test drives some of these functions from the `os` and `os.path` modules.

**Table 9.5. os Module File/Directory Access Functions**

| os *Module File/Directory Function* | *Operation* |
|---|---|
| ***File Processing*** | |
| `remove()/unlink()` | delete file |
| `rename()` | rename file |
| `*stat()` [a] | return file statistics |
| `symlink()` | create symbolic link |
| `utime()` | update timestamp |
| ***Directories/Folders*** | |
| `chdir()` | change working directory |
| `listdir()` | list files in directory |

| | |
|---|---|
| `getcwd()` | return current working directory |
| `mkdir()/makedirs()` | create directory(ies) |
| `rmdir()/removedirs()` | remove directory(ies) |
| ***Access/Permissions (available only on Unix U or Windows W)*** | |
| `access()` | verify permission modes U |
| `chmod()` | change permission modes UW |
| `umask()` | set default permission modes UW |

[a] includes `stat(), lstat(), xstat()`

**Table 9.6. `os.path` Module Pathname Access Functions**

| ***`os.path` Pathname Function*** | ***Operation*** |
|---|---|
| ***Separation*** | |
| `basename()` | remove directory path and return leaf name |
| `dirname()` | remove leaf name and return directory path |
| `join()` | join separate components into single pathname |
| `split()` | return (*`dirname(), basename()`*) tuple |
| `splitdrive()` | return (*`drivename, pathname`*) tuple |
| `splitext()` | return (*`filename, extension`*) tuple |
| ***Information*** | |
| `getatime()` | return last file access time |
| `getmtime()` | return last file modification time |
| `getsize()` | return file size (in bytes) |
| ***Inquiry*** | |
| `exists()` | does pathname (file or directory) exist? |
| `isdir()` | does pathname exist and is a directory? |
| `isfile()` | does pathname exist and is a file? |
| `islink()` | does pathname exist and is a symbolic link? |
| `samefile()` | do both pathnames point to the same file? |

### Example 9.1. `os & os.path` Modules Example (`ospathex.py`)

**This code exercises some of the functionality found in the** `os` and `os.path` modules. It creates a test file, populates a small amount of data in it, renames the file, and dumps its contents. Other auxiliary file operations are performed as well, mostly pertaining to directory tree traversal and file pathname manipulation.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import os
004 4  for tmpdir in ('/tmp', 'c:/windows/temp'):
005 5      if os.path.isdir(tmpdir):
006 6          break
007 7  else: <$nopage>
008 8       print 'no temp directory available'
009 9       tmpdir = ''
010 10
011 11 if tmpdir:
012 12     os.chdir(tmpdir)
013 13     cwd = os.getcwd()
014 14     print '*** current temporary directory'
015 15     print cwd
016 16
017 17     print '*** creating example directory…'
018 18     os.mkdir('example')
019 19     os.chdir('example')
020 20     cwd = os.getcwd()
021 21     print '*** new working directory:'
022 22     print cwd
023 23     print '*** original directory listing:'
024 24     print os.listdir(cwd)
025 25
026 26     print '*** creating test file…'
027 27     file = open('test', 'w')
028 28     file.write('foo\n')
029 29     file.write('bar\n')
030 30     file.close()
031 31     print '*** updated directory listing:'
032 32     print os.listdir(cwd)
033 33
034 34     print "*** renaming 'test' to 'filetest.txt'"
035 35     os.rename('test', 'filetest.txt')
036 36     print '*** updated directory listing:'
037 37     print os.listdir(cwd)
038 38
039 39     path = os.path.join(cwd, os.listdir (cwd)[0])
040 40     print '*** full file pathname'
041 41     print path
042 42     print '*** (pathname, basename) =='
043 43     print os.path.split(path)
044 44     print '*** (filename, extension) =='
045 45     print os.path.splitext(os.path.basename (path))
046 46
047 47     print '*** displaying file contents:'
048 48     file = open(path)
049 49     allLines = file.readlines()
050 50     file.close()
051 51     for eachLine in allLines:
052 52         print eachLine,
053 53
054 54     print '*** deleting test file'
055 55     os.remove(path)
056 56     print '*** updated directory listing:'
```

```
057 57     print os.listdir(cwd)
058 58     os.chdir(os.pardir)
059 59     print '*** deleting test directory'
060 60     os.rmdir('example')
061 61     print '*** DONE'
062  <$nopage>
```

Running this program on a Unix platform, we get the following output:

```
% ospathex.py
*** current temporary directory
/tmp
*** creating example directory…
*** new working directory:
/tmp/example
*** original directory listing:
[]
*** creating test file…
*** updated directory listing:
['test']
*** renaming 'test' to 'filetest.txt'
*** updated directory listing:
['filetest.txt']
*** full file pathname:
/tmp/example/filetest.txt
*** (pathname, basename) ==
('/tmp/example', 'filetest.txt')
*** (filename, extension) ==
('filetest', '.txt')
*** displaying file contents:
foo
bar
*** deleting test file
*** updated directory listing:
[]
*** deleting test directory
*** DONE
```

Running this example from a DOS window results in very similar execution:

```
C:\>python ospathex.py
*** current temporary directory
c:\windows\temp
*** creating example directory…
*** new working directory:
c:\windows\temp\example
*** original directory listing:
[]
*** creating test file…
*** updated directory listing:
['test']
*** renaming 'test' to 'filetest.txt'
```

```
*** updated directory listing:
['filetest.txt']
*** full file pathname:
c:\windows\temp\example\filetest.txt
*** (pathname, basename) ==
('c:\\windows\\temp\\example', 'filetest.txt')
*** (filename, extension) ==
('filetest', '.txt')
*** displaying file contents:
foo
bar
*** deleting test file
*** updated directory listing:
[]
*** deleting test directory
*** DONE
```

Rather than providing a line-by-line explanation here, we will leave it to the reader as an exercise. However, we will walk through a similar interactive example (including errors) to give you a feel for what it is like to execute this script one step at a time. We will break into the code every now and then to describe the code we just encountered.

```
>>> import os
>>> os.path.isdir('/tmp')
1
>>> os.chdir('/tmp')
>>> cwd = os.getcwd()
>>> cwd
'/tmp'
```

This first block of code consists of importing the `os` module (which also grabs the `os.path` module). We verify that `'/tmp'` is a valid directory and change to that temporary directory to do our work. When we arrive, we call the `getcwd()` method to tell us where we are.

```
>>> os.mkdir('example')
>>> cwd = os.getcwd()
>>> cwd
'/tmp/example'
>>>
>>> os.listdir()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: function requires at least one argument
>>>
>>> os.listdir(cwd)
[]
```

Next, we create a subdirectory in our temporary directory, after which we will use the `listdir()` method to confirm that the directory is indeed empty (since we just created it). The problem with our first call to `listdir()` was that we forgot to give the name of the directory we want to list. That problem is quickly remedied on the next line of input.

```
>>> file = open('test', 'w')
>>> file.write('foo\n')
>>> file.write('bar\n')
>>> file.close()
>>> os.listdir(cwd)
['test']
```

We then create a test file with two lines and verify that the file has been created by listing the directory again afterwards.

```
>>> os.rename('test', 'filetest.txt')
>>> os.listdir(cwd)
['filetest.txt']
>>>
>>> path = os.path.join(cwd, os.listdir(cwd)[0])
>>> path
'/tmp/example/filetest.txt'
>>>
>>> os.path.isfile(path)
1
>>> os.path.isdir(path)
0
>>>
>>> os.path.split(path)
('/tmp/example', 'filetest.txt')
>>>
>>> os.path.splitext(os.path.basename(path))
('filetest', '.ext')
```

This section is no doubt an exercise of `os.path` functionality, testing `join()`, `isfile()`, `isdir()` which we have seen earlier, `split()`, `basename()`, and `splitext()`. We also call the `rename()` function from `os`.

```
>>> file = open(path)
>>> file.readlines()
>>> file.close()
>>>
>>> for eachLine in allLines:
…       print eachLine,
…
foo
bar
```

This next piece of code should be familiar to the reader by now, since this is the third time around. We open the test file, read in all the lines, close the file, and display each line, one at a time.

```
>>> os.remove(path)
>>> os.listdir(cwd)
[]
>>> os.chdir(os.pardir)
>>> os.rmdir('example')
```

This last segment involves the deletion of the test file and test directory concluding execution. The call to `chdir()` moves us back up to the main temporary directory where we can remove the test directory (`os.pardir` contains the parent directory string `".."` for Unix and Windows; the Macintosh uses `"::"`). It is not advisable to remove the directory that you are in.

**NOTE**

As you can tell from our lengthy discussion above, the `os` and `os.path` modules provide different ways to access the file system on your computer. Although our study in this chapter is restricted to file access only, the `os` module can do much more. It lets you manage your process environment, contains provisions for low-level file access, allows you to create and manage new processes, and even enables your running Python program to "talk" directly to another running program. You may find yourself a common user of this module in no time. Read more about the `os` module in *Chapter 14*.

## File Execution

Whether we want to simply run an operating system command, invoke a binary executable, or another type of script (perhaps a shell script, Perl, or Tcl/Tk), this involves executing another file somewhere else on the system. Even running other Python code may call for starting up another Python interpreter, although that may not always be the case. In any regard, we will defer this subject to Chapter 14. Please proceed there if you are interested in how to start other programs, perhaps even communicating with them, and for general information regarding Python's execution environment.

## Persistent Storage Modules

In many of the exercises in this text, user input is required for those applications. After many iterations, it may be somewhat frustrating being required to enter the same data repeatedly. The same may occur if you are entering a significant amount of data for use in the future. This is where it becomes useful to have persistent storage, or a way to archive your data so that you may access it at a later time instead of having to re-enter all of that information. When simple disk files are no longer acceptable and full relational database management systems (RDBMSs) are overkill, simple persistent storage fills the

gap. The majority of the persistent storage modules deals with storing strings of data, but there are ways to archive Python objects as well.

## pickle and marshal Modules

Python provides a variety of modules which implement minimal persistent storage. One set of modules (marshal and pickle) allows for pickling of Python objects. Pickling is the process whereby objects more complex than primitive types can be converted to a binary set of bytes that can be stored or transmitted across the network, then be converted back to their original object forms. Pickling is also known as flattening, serializing, or marshalling. Another set of modules (dbhash/bsddb, dbm, gdbm, dumbdbm) and their "manager" (anydbm) can provide persistent storage of Python strings only. The last module (shelve) can do both.

As we mentioned before, both marshal and pickle can flatten Python objects. These modules do not provide "persistent storage" per se, since they do not provide a namespace for the objects, nor can they provide concurrent write access to persistent objects. What they can do, however, is to pickle Python objects to allow them to be stored or transmitted. Storage, of course, is sequential in nature (you store or transmit objects one after another). The difference between marshal and pickle is that marshal can handle only simple Python objects (numbers, sequences, mapping, and code) while pickle can transform recursive objects, objects that are multi-referenced from different places, and user-defined classes and instances. The pickle module is also available in a turbo version called cPickle, which implements all functionality in C.

## DBM-style Modules

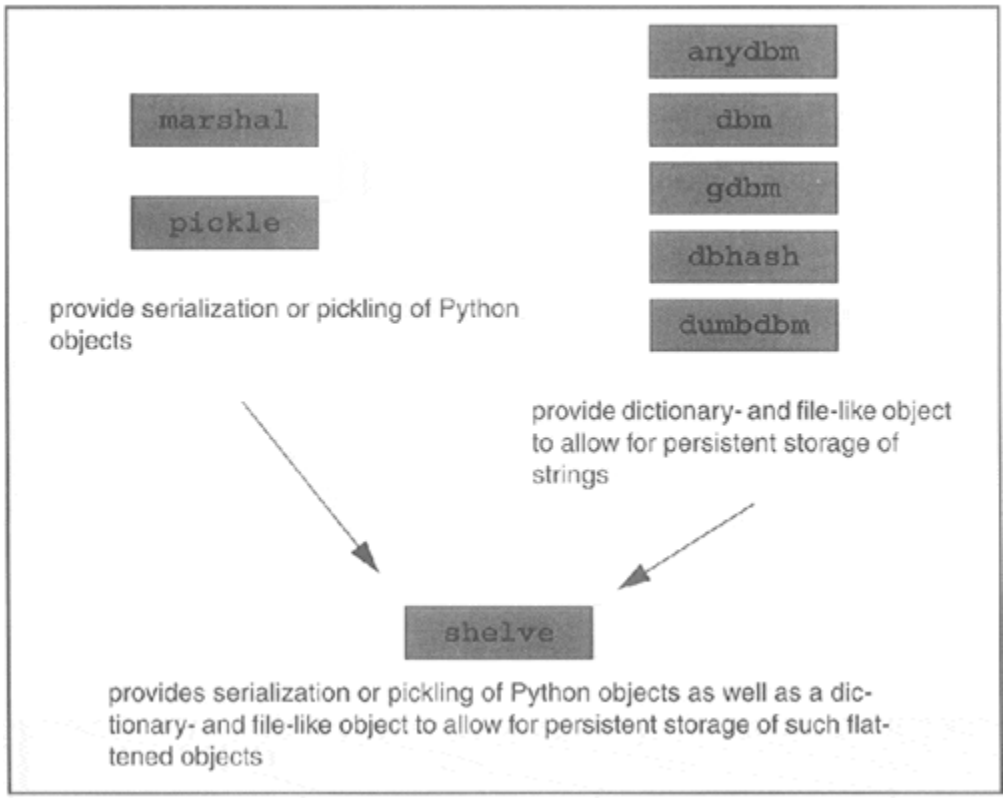
The *db* series of modules writes data in the traditional DBM format. There are a large number of different implementations: dbhash/bsddb, dbm, gdbm, and dumbdbm. We highly recommend the use of the anydbm module, which detects which DBM-compatible modules are installed on your system and uses the "best" one at its disposal. The dumbdbm module is the most limited one, and is the default used if none of the other packages are available. These modules do provide a namespace for your objects, using objects which behave similar to a combination of a dictionary object and a file object. The one limitation of these systems is that they can store only strings. In other words, they do not serialize Python objects.

## shelve Module

Finally, we have a somewhat more complete solution, the shelve module. The shelve module uses the anydbm module to find a suitable DBM module, then uses cPickle to perform the pickling process. The shelve module permits concurrent read access to the database file, but not shared read/write access. This is about as close to persistent storage as you will find in the Python standard library. There may other external extension modules which implement "true" persistent storage. The diagram in Figure 9-1 shows the

relationship between the pickling modules and the persistent storage modules, and how the shelve object appears to be the best of both worlds.

**Figure 9.1. Python Modules for Serialization and Persistency**




**NOTE**

The `pickle` module allows you to store Python objects directly to a file without having to convert them to strings or to necessarily write them out as binary files using low-level file access. Instead, the `pickle` module creates a Python-only binary version which allows you to cleanly read and write objects in their entirety without having to worry about all the file details. All you need is a valid file handle, and you are ready to read or write objects from or to disk.

The two main functions in the `pickle` module are `dump()` and `load()`. The `dump()` function takes a file handle and a data object and saves the object in a format it understands to the given file. When a pickled object is loaded from disk using `load()`, it knows exactly how to restore that object to its original configuration before it was saved to disk. We recommend you take a look at `pickle` and its "smarter" brother `shelve`, which gives you dictionary-like functionality so there is even less file overhead on your part.

## Related Modules

There are plenty of other modules related to files and input/output, all of which work on most of the major platforms. Table9.7 lists some of the file-related modules.

**Table 9.7. Related File Modules**

| *Module(s)* | *Contents* |
|---|---|
| fileinput | iterates over lines of multiple input text files |
| getopt | provides command-line argument parsing/manipulation |
| glob/fnmatch | provides Unix-style wildcard character matching |
| gzip/zlib/zipfile[a] | allows file access to include automatic de/compression |
| shutil | offers high-level file access functionality |
| c/StringIO | implements file-like interface on top of string objects |
| tempfile | generates temporary file names or files |

[a] new in Python 1.6

The `fileinput` module iterates over a set of input files and reads their contents one line at a time, allowing you to iterate over each line, much like the way the Perl ( `< >` ) operator works without any provided arguments. File names that are not explicitly given will be assumed to be provided from the command-line.

The `glob` and `fnmatch` modules allow for file name pattern-matching in the good old fashioned Unix shell-style, for example, using the asterisk ( `*` ) wildcard character for all string matches and the ( `?` ) for matching single characters.

The `gzip` and `zlib` modules provide direct file access to the `zlib` compression library. The `gzip` module, written on top of the `zlib` module, allows for standard file access, but provides for automatic `gzip`-compatible compression and decompression. Note that if you are compiling your Python interpreter, you have to enable the `zlib` module to be built (by editing the `Modules/Setup` file). It is not turned on by default. The new `zipfile` module, also requiring the `zlib` module, allows the programmer to create, modify, and read `zip` archive files.

The `shutil` module furnishes high-level file access, performing such functions as copying files, copying file permissions, and recursive directory tree copying, to name a few.

The `tempfile` module can be used to generate temporary file names and files.

In our earlier chapter on strings, we described the `StringIO` module (and its C-compiled companion `cStringIO`), and how it overlays a file interface on top of string objects. This interface includes all of the standard methods available to regular file objects.

The modules we mentioned in the Persistent Storage section above (Section 9.9) include examples of a hybrid file- and dictionary-like object.

Some other Python modules which generate file-like objects include network and file socket objects (`socket` module), the `popen*()` file objects that connect your application

to other running processes (`os` and `popen2` modules), the `fdopen()` file object used in low-level file access (`os` module), and opening a network connection to an Internet web server via its Uniform Resource Locator (URL) address (`urllib` module). Please be aware that not all standard file methods may be implemented for these objects. Likewise, they may provide functionality in addition to what is available for regular files.

Refer to the documentation for more details on these file access-related modules.

## Exercises

**1:**

*File Filtering.* Display all lines of a file, except those that start with a pound sign ( `#` ), the comment character for Python, Perl, Tcl, and most other scripting languages.

**2:**

*File Access.* Prompt for a number *N* and file *F,* and display the first *N* lines of *F*.

**3:**

*File Information.* Prompt for a filename and display the number of lines in that text file.

**4:**

*File Access.* Write a "pager" program. Your solution should prompt for a file name, and display the text file 25 lines at a time, pausinLg each time to ask the user to "press a key to continue."

**5:**

*Test Scores.* Update your solution to the test scores problems (Exercises 5–3 and 6–4) by allowing a set of test scores be loaded from a file. We leave the file format to your discretion.

**6:**

*File Comparison.* Write a program to compare two text files. If they are different, give the line and column numbers in the files where the first difference occurs.

**7:**

*Parsing Files.* Windows users: create a program that parses a Windows `.ini` file. Unix users: create a program that parses the `/etc/services` file. All other platforms: create a program that parses an system file with some kind of structure

to it.

**8:**

*Module Introspection.* Extract module attribute information. Prompt the user for a module name (or accept it from the command-line). Then, using `dir()` and other built-in functions, extract all its attributes, and display their names, types, and values.

**9:**

"*PythonDoc.*" Go to the directory where your Python standard library modules are located. Examine each `.py` file and determine whether a `__doc__` string is available for that module. If so, format it properly and catalog it. When your program has completed, it should present a nice list of those modules which have documentation strings and what they are. There should be a trailing list showing which modules do not have documentation strings (the shame list). EXTRA CREDIT: extract documentation for all classes and functions within the standard library modules.

**10:**

*Home Finances.* Create a home finance manager. Your solution should be able to manage savings, checking, money market, certificate of deposit (CD), and similar accounts. Provide a menu-based interface to each account as well as operations such as deposits, withdrawals, debits, and credits. An option should be given to a user to remove transactions as well. The data should be stored to file when the user quits the application (but randomly during execution for backup purposes).

**11:**

Web Site Addresses.

(a)Write a URL bookmark manager. Create a text-driven menu-based application which allows the user to add, update, or delete entries. Entries include a site name, Website URL address, and perhaps a one-line description (optional). Allow search functionality so that a search "word" looks through both names and URLs for possible matches. Store the data to a disk file when the user quits the application, and load up the data when the user restarts.

(b)Upgrade your solution to part (a) by providing output of the bookmarks to a legible and syntactically correct HTML file (`.htm` or `.html`) so that users can then point their browser to this output file and be presented with a list of their bookmarks. Another feature to implement is allowing the creation of "folders" to allow grouping of related bookmarks. EXTRA CREDIT: Read the literature on regular expressions and the Python `re` module. Add regular expression validation

of URLs that users enter into their database.

**12:**

Users and Passwords.

(a)Do Exercise 7-5, which keeps track of usernames and passwords. Update your code to support a "last login time." See the documentation for the time module to obtain timestamps for when users "login" to the system. Also, create the concept of an "administrative" user which can dump a list of all the users, their passwords (you can add encryption on top of the passwords if you wish), and their last login times. The database should be stored to disk, one line at a time, with fields delimited by colons ( `:` ), i.e., "`joe:boohoo:953176591.145,`" for each user. The number of lines in the file will be the number of users which are part of your system.

(b)Further update your example such that instead of writing out one line at a time, you "pickle" the entire database object and write that out instead. Read the documentation on the `pickle` module to find out how to "flatten" or "serialize" your object, as well as how to perform I/O using picked objects. With the addition of this new code, your solution should take up fewer lines than your solution in part (a).

**13:**

Command-line arguments.

(a)What are they, and why might they be useful?

(b)Write code to display the command-line arguments which were entered.

**14:**

*Logging Results.* Convert your calculator program (Exercise 5-6) to take input from the command-line, i.e.,

```
% calc.py 1 + 2
```

Output the result only. Also, write each expression and result to a disk file. Issuing a command of…

```
% calc.py print
```

… will cause the entire contents of the "register tape" to be dumped to the screen and file reset/truncated. Here is an example session:

```
% calc.py 1 + 2
3
% calc.py 3 ^ 3
27
% calc.py print
1 + 2
3
3 ^ 3
27
% calc.py print
%
```

**15:**

*Copying Files.* Prompt for two file names (or better yet, use command-line arguments). The contents of the first file should be copied to the second file.

**16:**

*Text Processing.* You are tired of seeing lines on your e-mail wrap because people type lines which are too long for your mail reader application. Create a program to scan a text file for all lines longer than 80 characters. For each of the offending lines, find the closest word before 80 characters and break the line there, inserting the remaining text to the next line (and pushing the previous next line down one). When you are done, there should no longer be lines longer than 80 characters.

**17:**

*Text Processing.* Create a crude and elementary text file editor. Your solution is menu-driven, with the following options: (1) create file [prompt for file name and any number of lines of input], (2) display file [dump its contents to the screen], (3) edit file (prompt for line to edit and allow user to make changes), (4) save file, and (5) quit.

**18:**

*Searching Files.* Obtain a byte value (0-255) and a file name. Display the number of times that byte appears in the file.

**19:**

*Generating Files.* Create a sister program to the previous problem. Create a binary data file with random bytes, but one particular byte will appear in that file a set number of times. Obtain the following three values: (1) a byte value (0–255), (2) the number of times that byte should appear in the data file, and (3) the total

number of bytes that make up the data file. Your job is to create that file, randomly scatter the request byte across the file, and to ensure that there are no duplicates and that the file contains exactly the number of occurrences that byte was requested for, and that the resulting data file is exactly the size requested.

# Chapter 10. Errors And Exceptions

Errors are an everyday occurrence in the life of a programmer. In days hopefully long since past, errors were either fatal to the program (or perhaps the machine) or produced garbage output that was neither recognized as valid input by other computers or programs nor by the humans who submitted the job to be run. Any time an error occurred, execution was halted until the error was corrected and code was re-executed. Over time, demand surged for a "softer" way of dealing with errors other than termination. Programs evolved such that not every error was malignant, and when they did happen, more diagnostic information was provided by either the compiler or the program during run-time to aid the programmer in solving the problem as quickly as possible. However, errors are errors, and any resolution usually took place after the program or compilation process was halted. There was never really anything a piece of code could do but exit and perhaps leave some crumbs hinting at a possible cause—until *exceptions* and *exception handling* came along.

Although we have yet to cover classes and object-oriented programming in Python, many of the concepts presented here involve classes and class instances.[1] We conclude the chapter with an optional section on how to create your own exception classes. Older versions of Python utilized string exceptions, which are not common any more. We recommend using only class-based exceptions for all future development.

[1] As of Python 1.5, all standard exceptions are implemented as classes. If new to classes, instances, and other object-oriented terminology, the reader should check Chapter 13 for clarification.

This chapter begins by exposing the reader to exceptions, exception handling, and how they are supported in Python. We also describe how programmers can generate exceptions within their code. Finally, we reveal how programmers can create their own exception classes.

## What Are Exceptions?

### Errors

Before we get into details about what exceptions are, let us review what errors are. In the context of software, errors are either syntactical or logical in nature. Syntax errors indicate errors with the construct of the software and cannot be executed by the interpreter or compiled correctly. These errors must be repaired before execution can occur.

Once programs are semantically correct, the only errors which remain are logical. Logical errors can either be caused by lack of or invalid input, or, in other cases, by the logic's not

being able to generate, calculate, or otherwise produce the desired results based on the input. These errors are sometimes known as domain and range failures, respectively.

When errors are detected by Python, the interpreter indicates that it has reached a point where continuing to execute in the current flow is no longer possible. This is where exceptions come into the picture.

## Exceptions

Exceptions can best be described as action that is taken outside of the normal flow of control because of errors. This action comes in two distinct phases, the first being the error which causes an exception to occur, and the second being the detection (and possible resolution) phase.

The first phase takes place when an *exception condition* (sometimes referred to as *exceptional condition*) occurs. Upon detection of an error and recognition of the exception condition, the interpreter performs an operation called *raising* an exception. Raising is also known as triggering, throwing, or generating, and is the process whereby the interpreter makes it known to the current control flow that something is wrong. Python also supports the ability of the programmer's to raise exceptions. Whether triggered by the Python interpreter or the programmer, exceptions signal that an error has occurred. The current flow of execution is interrupted to process this error and take appropriate action, which happens to be the second phase.

The second phase is where exception handling takes place. Once an exception is raised, a variety of actions can be invoked in response to that exception. These can range anywhere from ignoring the error, logging the error but otherwise taking no action, performing some corrective measures and aborting the program, or alleviating the problem to allow for resumption of execution. Any of these actions represents a *continuation,* or an alternative branch of control. The key is that the programmer can dictate how the program operates when an error occurs.

As you may have already concluded, errors during run-time are primarily caused by external reasons, such as poor input, a failure of some sort, etc. These causes are not under the direct control of the programmer, who can anticipate only a few of the errors and code the most general remedies.

Languages like Python which support the raising and—more importantly—the handling of exceptions empowers the developer by placing them in a more direct line of control when errors occur. The programmer not only has the ability to detect errors, but also to take more concrete and remedial actions when they occur. Due to the ability to manage errors during run-time, application robustness is increased.

Exceptions and exception handling are not new concepts, as they are also present in Ada, Modula-3, C++, Eiffel, and Java. The origins of exceptions probably come from operating systems code which handles exceptions such as system errors and hardware interruptions. Exception handling as a software tool made its debut in the mid-1960s with

PL/1 being the first major programming language that featured exceptions. Like some of the other languages supporting exception handling, Python is endowed with the concepts of a "try" block and "catching" exceptions and, in addition, provides for more "disciplined" handling of exceptions. By this we mean that you can create different handlers for different exceptions, as opposed to a general "catch-all" code where you may be able to detect the exception which occurred in a post-mortem fashion.

## Exceptions in Python

As you were going through some of the examples in the previous chapters, you no doubt noticed what happens when your program "crashes" or terminates due to unresolved errors. A "traceback" notice appears along with a notice with as much diagnostic information as the interpreter can give you, including the error name, reason, and perhaps even the line number near or exactly where the error occurred. All errors have a similar format, regardless of whether running within the Python interpreter or standard script execution, providing a consistent error interface. All errors, whether they be syntactical or logical, result from behavior incompatible with the Python interpreter and cause exceptions to be raised.

Let us take a look at some exceptions now.

### `NameError:` attempt to access an undeclared variable

```
>>> foo
Traceback (innermost last):
 File "<interactive input>", line 0, in ?
NameError: foo
```

`NameError` indicates access to an uninitialized variable. The offending identifier was not found in the Python interpreter's symbol table. We will be discussing *namespaces* in an upcoming chapter, but as an introduction, regard them as "address books" linking names to objects. Any object which is accessible should be listed in a namespace. Accessing a variable entails a search by the interpreter, and if the name requested is not found in any of the namespaces, a `NameError` exception will be generated.

### `ZeroDivisionError:` division by any numeric zero

```
>>> 12.4/0.0
Traceback (innermost last):
 File "<interactive input>", line 0, in ?
ZeroDivisionError: float division
```

Our example above used floats, but in general, any numeric division-by-zero will result in a `ZeroDivisionError` exception.

### `SyntaxError:` Python interpreter syntax error

```
>>> for
  File "<string>", line 1
    for
       ^
SyntaxError: invalid syntax
```

`SyntaxError` exceptions are the only ones which do not occur at run-time. They indicate an improperly constructed piece of Python code which cannot execute until corrected. These errors are generated at compile-time, when the interpreter loads and attempts to convert your script to Python bytecode. These may also occur as a result of importing a faulty module.

### `IndexError:` request for an out-of-range index for sequence

```
>>> aList = []
>>> aList[0]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

`IndexError` is raised when attempting to access an index which is outside the valid range of a sequence.

### `KeyError:` request for a non-existent dictionary key

```
>>> aDict = {'host': 'earth', 'port': 80}
>>> print aDict['server']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: server
```

Mapping types such as dictionaries depend on keys to access data values. Such values are not retrieved if an incorrect/nonexistent key is requested. In this case, a `KeyError` is raised to indicate such an incident has occurred.

### `IOError:` input/output error

```
>>> f = open("blah")
Traceback (innermost last):
 File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'blah'
```

Attempting to open a non-existent disk file is one example of an operating system input/output (I/O) error. Any type of I/O error raises an `IOError` exception.

### `AttributeError:` attempt to access an unknown object attribute

```
>>> class myClass:
…     pass
…
>>> myInst = myClass()
>>> myInst.bar = 'spam'
>>> myInst.bar
'spam'
>>> myInst.foo
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: foo
```

In our example, we stored a value in `myInst.bar`, the `bar` attribute of instance `myInst`. Once an attribute has been defined, we can access it using the familiar dotted-attribute notation, but if it has not, as in our case with the `foo` (non-)attribute, an `AttributeError` occurs.

## Detecting and Handling Exceptions

Exceptions can be detected by incorporating them as part of a **`try`** statement. Any code suite of a **`try`** statement will be monitored for exceptions.

There are two main forms of the **`try`** statement: **`try-except`** and **`try-finally`**. These statements are mutually exclusive, meaning that you pick only one of them. A **`try`** statement is either accompanied by one or more **`except`** clauses or exactly one **`finally`** clause. (There is no such thing as a hybrid "`try-except-finally`.")

**`try-except`** statements allow one to detect and handle exceptions. There is even an optional `else` clause for situations where code needs to run only when no exceptions are detected. Meanwhile, **`try-finally`** statements allow only for detection and processing of any obligatory clean-up (whether or not exceptions occur), but otherwise has no facility in dealing with exceptions.

### `try-except` Statement

The **try-except** statement (and more complicated versions of this statement) allows you to define a section of code to monitor for exceptions and also provides the mechanism to execute handlers for exceptions.

The syntax for the most general **try-except** statement looks like this:

```
try:
    try_suite        # watch for exceptions here
except
                                                    Exception:
    except_suite     # exception-handling code
```

Let us give one example, then explain how things work. We will use our `IOError` example from above. We can make our code more robust by adding a **try-except** "wrapper" around the code:

```
>>> try:
…       f = open('blah')
…   except IOError:
…       print 'could not open file'
…
could not open file
```

As you can see, our code now runs seemingly without errors. In actuality, the same `IOError` still occurred when we attempted to open the nonexistent file. The difference? We added code to both detect and handle the error. When the `IOError` exception was raised, all we told the interpreter to do was to output a diagnostic message. The program continues and does not "bomb out" as our earlier example—a minor illustration of the power of exception handling. So what is really happening codewise?

During run-time, the interpreter attempts to execute all the code within the **try** statement. If an exception does not occur when the code block has completed, execution resumes past the **except** statement. When the specified exception named on the **except** statement does occur, control flow immediately continues in the handler (all remaining code in the `try` clause is skipped). In our example above, we are catching only `IOError` exceptions. Any other exception will not be caught with the handler we specified. If, for example, you want to catch an `OSError`, you have to add a handler for that particular exception. We will elaborate on the **try-except** syntax more as we progress further in this chapter.

**NOTE**

The remaining code in the **try** suite from the point of the exception is never reached (hence never executed). Once an exception is raised, the race is on to decide on the

continuing flow of control. The remaining code is skipped, and the search for a handler begins. If one is found, the program continues in the handler.

If the search is exhausted without finding an appropriate handler, the exception is then propagated to the caller's level for handling, meaning the stack frame immediately preceding the current one. If there is no handler at the next higher level, the exception is yet again propagated to its caller. If the top level is reached without an appropriate handler, the exception is considered unhandled, and the Python interpreter will display the traceback and exit.

## Wrapping a Built-in Function

We will now present an interactive example—starting with the bare necessity of detecting an error, then building continuously on what we have to further improve the robustness of our code. The premise is in detecting errors while trying to convert a numeric string to a proper (numeric object) representation of its value.

The `float()` built-in function has a primary purpose of converting any numeric type to a float. In Python 1.5, `float()` was given the added feature of being able to convert a number given in string representation to an actual float value, obsoleting the use of the `atof()` function of the `string` module. Readers with older versions of Python may still use `string.atof()`, replacing `float()`, in the examples we use here.

```
>>> float(12345)
12345.0
>>> float('12345')
12345.0
>>> float('123.45e67')
1.2345e+069
```

Unfortunately, `float()` is not very forgiving when it comes to bad input:

```
>>> float('abcde')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    float('abcde')
ValueError: invalid literal for float(): abcde
>>>
>>> float(['this is', 1, 'list'])
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    float(['this is', 1, 'list'])
TypeError: object can't be converted to float
```

Notice in the errors above that `float()` does not take too kindly to strings which do not represent numbers or non-strings. Specifically, if the correct argument type was given (string type) but that type contained an invalid value, the exception raised would be `ValueError` because it was the value that was improper, not the type. In contrast, a list is a bad argument altogether, not even being of the correct type; hence, `TypeError` was thrown.

Our exercise is to call `float()` "safely," or in a more "safe manner," meaning that we want to ignore error situations because they do not apply to our task of converting numeric string values to floating point numbers, yet are not severe enough errors that we feel the interpreter should abandon execution. To accomplish this, we will create a "wrapper" function, and, with the help of **`try-except`**, create the environment that we envisioned. We shall call it `safe_float()`. In our first iteration, we will scan and ignore only `ValueErrors`, because they are the more likely culprit. `TypeErrors` rarely happen since somehow a non-string must be given to `float()`.

```
def safe_float(object):
     try:
         return float(object)
     except ValueError:
         pass
```

The first step we take is to just "stop the bleeding." In this case, we make the error go away by just "swallowing it." In other words, the error will be detected, but since we have nothing in the **`except`** suite (except the **`pass`** statement, which does nothing but serve as a syntactical placeholder for where code is supposed to go), no handling takes place. We just ignore the error.

One obvious problem with this solution is that we did not explicitly return anything to the function caller in the error situation. Even though `None` is returned (when a function does not return any value explicitly, i.e., completing execution without encountering a **`return`** *`object`* statement), we give little or no hint that anything wrong took place. The very least we should do is to explicitly return `None` so that our function returns a value in both cases and makes our code somewhat easier to understand:

```
def safe_float(object):
    try:
         retval = float(object)
    except ValueError:
         retval = None
    return retval
```

Bear in mind that with our change above, nothing about our code changed except that we used one more local variable. In designing a well-written application programmer interface (API), you may have kept the return value more flexible. Perhaps you documented that if a proper argument was passed to `safe_float()`, then indeed, a floating point number would be returned, but in the case of an error, you chose to return a string indicating the problem with the input value. We modify our code one more time to reflect this change:

```
def safe_float(object):
     try:
         retval = float(object)
     except ValueError:
         retval = 'could not convert non-number to float'
     return retval
```

The only thing we changed in the example was to return an error string as opposed to just `None`. We should take our function out for a "test drive" to see how well it works so far:

```
>>> safe_float('12.34')
12.34
>>> safe_float('bad input')
'could not convert non-number to float'
```

We made a good start—now we can detect invalid string input, but we are still vulnerable to invalid *objects* being passed in:

```
>>> safe_float({'a': 'Dict'})
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "safeflt.py", line 28, in safe_float
    retval = float(object)
TypeError: object can't be converted to float
```

We will address this final shortcoming momentarily, but before we further modify our example, we would like to highlight the flexibility of the **`try-except`** syntax, especially the **`except`** statement, which comes in a few more flavors.

## `try` Statement with Multiple `excepts`

Earlier in this chapter, we introduced the following general syntax for **`except:`**

```
except
```

```
                                                    Exception:
    suite_for_exception_Exception
```

The **except** statement in such formats specifically detects exceptions named *Exception*. You can chain multiple **except** statements together to handle different types of exceptions with the same **try:**

```
except
                                                    Exception1:
    suite_for_exception_Exception1
except
                                                    Exception2:
    suite_for_exception_Exception2
                :
```

This same **try** clause is attempted, and if there is no error, execution continues, passing all the **except** clauses. However, if an exception *does* occur, the interpreter will look through your list of handlers attempting to match the exception with one of your handlers (**except** clauses). If one is found, execution proceeds to *that* **except** suite.

Our `safe_float()` function has some brains now to detect specific exceptions. Even smarter code would handle each appropriately. To do that, we have to have separate **except** statements, one for each exception type. That is no problem as Python allows `except` statements can be chained together. Any reader familiar with popular third-generation languages (3GLs) will no doubt notice the similarities to the `switch/case` statement which is absent in Python. We will now create separate messages for each error type, providing even more detail to the user as to the cause of his or her problem:

```
def safe_float(object):
    try:
        retval = float(object)
    except ValueError:
        retval = 'could not convert non-number to float'
    except TypeError:
        retval = 'object type cannot be converted to float'
    return retval
```

Running the code above with erroneous input, we get the following:

```
>>> safe_float('xyz')
'could not convert non-number to float'
>>> safe_float(())
'argument must be a string'
>>> safe_float(200L)
```

```
200.0
>>> safe_float(45.67000)
45.67
```

## `except` Statement with Multiple Exceptions

We can also use the same **`except`** clause to handle multiple exceptions. **`except`** statements which process more than one exception require that the set of exceptions be contained in a tuple:

```
except (Exception1, Exception2):
     suite_for_Exception1_and_Exception2
```

The above syntax example illustrates how two exceptions can be handled by the same code. In general, any number of exceptions can follow an **`except`** statement as long as they are all properly enclosed in a tuple:

```
except (Exception1[, Exception2[, … ExceptionN…]]):
     suite_for_exceptions_Exception1_to_ExceptionN
```

If for some reason, perhaps due to memory constraints or dictated as part of the design that all exceptions for our `safe_float()` function must be handled by the same code, we can now accommodate that requirement:

```
def safe_float(object):
    try:
        retval = float(object)
    except (ValueError, TypeError):
        retval = 'argument must be a number or numeric string'
    return retval
```

Now there is only the single error string returned on erroneous input:

```
>>> safe_float('Spanish Inquisition')
'argument must be a number or numeric string'
>>> safe_float([])
'argument must be a number or numeric string'
>>> safe_float('1.6')
1.6
>>> safe_float(1.6)
1.6
```

```
>>> safe_float(932)
932.0
```

## try-except with No Exceptions Named

The final syntax for **try-except** we are going to present is one which does not specify an exception on the except header line:

```
try:
    try_suite      # watch for exceptions here
except:
    except_suite   # handles all exceptions
```

Although this code "catches the most exceptions," it does not promote good Python coding style. One of the chief reasons is that it does not take into account the potential root causes of problems which may generate exceptions. Rather than investigating and discovering what types of errors may occur and how they may be prevented from happening, this type of code "turns the blind eye," thereby ignoring the possible causes (and remedies). Also see the Core Style featured in this section.

**NOTE**

The **try-except** statement has been included in Python to provide a powerful mechanism for programmers to track down potential errors and to perhaps provide logic within the code to handle situations where it may not otherwise be possible, for example in C. The main idea is to minimize the number of errors and still maintain program correctness. As with all tools, they must be used properly.

One incorrect use of **try-except** is to serve as a giant bandage over large pieces of code. By that we mean putting large blocks, if not your entire source code, within a **try** and/or have a large generic **except** to "filter" any fatal errors by ignoring them:

```
# this is really bad code
try:

    large_block_of_code # bandage of large piece of code
except:
                                                        pass
# blind eye ignoring all errors
```

Obviously, errors cannot be avoided, and the job of **try-except** is to provide a mechanism whereby an acceptable problem can be remedied or properly dealt with, and

not be used as a filter. The construct above will hide many errors, but this type of usage promotes a poor engineering practice that we certainly cannot endorse.

Bottom line: Avoid using **try-except** around a large block of code with a **pass** just to hide errors. Instead, handle specific exceptions and enclose only deserving code in your **try** clause, as evidenced by some of the constructs we used for the `safe_float()` example in this section.

## "Exceptional Arguments"

No, the title of this section has nothing to do with having a major fight. Instead, we are referring to the fact that exception may have *arguments* are passed along to the exception handler when they are raised. When an exception is raised, parameters are generally provided as an additional aid for the exception handler. Although arguments to exceptions are optional, the standard built-in exceptions do provide at least one argument, an error string indicating the cause of the exception.

Exception parameters can be ignored in the handler, but the Python provides syntax for saving this value. To access any provided exception argument, you must reserve a variable to hold the argument. This argument is given on the **except** header line and follows the exception type you are handling. The different syntaxes for the **except** statement can be extended to the following:

```
# single exception
except
                                                    Exception,
Argument:
     suite_for_Exception_with_Argument

# multiple exceptions
except (Exception1, Exception2, …, ExceptionN), Argument:
     suite_for_Exception1_to_ExceptionN_with_Argument
```

Unless a string exception (see Section 10.4) was raised, *Argument* is a class instance containing diagnostic information from the code raising the exception. The exception arguments themselves go into a tuple which is stored as an attribute of the class instance, an instance of the exception class from which it was instantiated. In the first alternate syntax above, *Argument* would be an instance of the `Exception` class.

For most standard built-in exceptions, that is, exceptions derived from `StandardError`, the tuple consists of a single string indicating the cause of the error. The actual exception name serves as a satisfactory clue, but the error string enhances the meaning even more. Operating system or other environment type errors, i.e., `IOError`, will also include an

operating system error number which precedes the error string in the tuple. Whether an *Argument* is merely a string or a combination of an error number and a string, calling str(*Argument*) should present a human-readable cause of an error.

The only caveat is that not all exceptions raised in third-party or otherwise external modules adhere to this standard protocol (or error string or (error number, error string). We recommend to follow such a standard when raising your own exceptions (see Core Style note).

**NOTE**

When you raise built-in exceptions in your own code, try to follow the protocol established by the existing Python code as far as the error information that is part of the tuple passed as the exception argument. In other words, if you raise a ValueError, provide the same argument information as when the interpreter raises a ValueError exception, and so on. This helps keep the code consistent and will prevent other code which uses your module from breaking.

The example below is when an invalid object is passed to the float() built-in function, resulting in a TypeError exception:

```
>>> try:
…       float(['float() does not', 'like lists', 2])
…   except TypeError, diag:# capture diagnostic info
…       pass
…
>>> type(diag)
<type 'instance'>
>>>
>>> print diag
object can't be converted to float
```

The first thing we did was cause an exception to be raised from within the **try** statement. Then we passed cleanly through by ignoring but saving the error information. Calling the type() built-in function, we were able to confirm that our exception was indeed an instance. Finally, we displayed the error by calling print with our diagnostic exception argument.

To obtain more information regarding the exception, we can use the special __class__ instance attribute which identifies which class an instance was instantiated from. Class objects also have attributes, such as a documentation string and a string name which further illuminate the error type:

```
>>> diag                        # exception instance object
<exceptions.TypeError instance at 8121378>
>>> diag.__class__              # exception class object
<class exceptions.TypeError at 80f6d50>
>>> diag.__class__.__doc__     # exception class documentation  string
'Inappropriate argument type.'
>>> diag.__class__.__name__    # exception class name
'TypeError'
```

As we will discover in Chapter 13—Classes and OOP—the special instance attribute `__class__` exists for all class instances, and the `__doc__` class attribute is available for all classes which define their documentation strings.

We will now update our `safe_float()` one more time to include the exception argument which is passed from the interpreter from within `float()` when exceptions are generated. In our last modification to `safe_float()`, we merged both the handlers for the `ValueError` and `TypeError` exceptions into one because we had to satisfy some requirement. The problem, if any, with this solution is that no clue is given as to which exception was raised nor what caused the error. The only thing returned is an error string which indicated some form of invalid argument. Now that we have the exception argument, this no longer has to be the case.

Because each exception will generate its own exception argument, if we chose to return this string rather than a generic one we made up, it would provide a better clue as to the source of the problem. In the following code snippet, we replace our single error string with the string representation of the exception argument.

```
def safe_float(object):
     try:
         retval = float(object)
     except (ValueError, TypeError), diag:
         retval = str(diag)
    return retval
```

Upon running our new code, we obtain the following (different) messages when providing improper input to `safe_float()`, even if both exceptions are managed by the same handler:

```
>>> safe_float('xyz')
'invalid literal for float(): xyz'
>>> safe_float({})
'object can't be converted to float'
```

## Using Our Wrapped Function in an Application

We will now feature `safe_float()` in a mini application which takes a credit card transaction data file (`carddata.txt`) and reads in all transactions, including explanatory strings. Here are the contents of our example `carddata.txt` file:

```
% cat carddata.txt
# carddata.txt
previous balance
25
debits
21.64
541.24
25
credits
-25
-541.24
finance charge/late fees
7.30
5
```

Our program, `cardrun.py`, is given in Example 10.1.

### Example 10.1. Credit Card Transactions (`cardrun.py`)

We use `safe_float()` to process a set of credit card transactions given in a file and read in as strings. A log file tracks the processing.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import types
004 4
005 5  def safe_float(object):
006 6      'safe version of float()'
007 7      try: <$nopage>
008 8          retval = float(object)
009 9      except (ValueError, TypeError), diag:
010 10         retval = str(diag)
011 11     return retval
012 12
013 13 def main():
014 14     'handles all the data processing'
015 15     log = open('cardlog.txt', 'w')
016 16     try: <$nopage>
017 17         ccfile = open('carddata.txt', 'r')
018 18     except IOError:
019 19         log.write('no txns this month\n')
020 20         log.close()
021 21         return <$nopage>
022 22
```

```
023 23     txns = ccfile.readlines()
024 24     ccfile.close()
025 25     total = 0.00
026 26     log.write('account log:\n')
027 27
028 28     for eachTxn in txns:
029 29         result = safe_float(eachTxn)
030 30         if type(result) == types.FloatType:
031 31             total = total + result
032 32             log.write('data… processed\n')
033 33         else: <$nopage>
034 34             log.write('ignored: %s' % result)
035 35     print '$%.2f (new balance)' % (total)
036 36     log.close()
037 37
038 38 if __name__ == '__main__':
039 39     main()
040  <$nopage>
```

**Lines 1 – 3**

The script starts by importing the `types` modules, which contains Type objects for the Python types. That is why we direct them to standard error instead.

**Lines 5 – 11**

This chunk of code contains the body of our `safe_float()` function.

**Lines 13 – 36**

The core part of our application performs three major tasks: (1) read the credit card data file, (2) process the input, and (3) display the result. Lines 16–24 perform the extraction of data from the file. You will notice that there is a **`try-except`** statement surrounding the file open.

A log file of the processing is also kept. In our example, we are assuming the log file can be opened for write without any problems. You will find that our progress is kept by the log. If the credit card data file cannot be accessed, we will assume there are no transactions for the month (lines 18–21).

The data is then read into the `txns` (transactions) list where it is iterated over in lines 28–34. After every call to `safe_float()`, we check the result type using the `types` module. The `types` module contains items of each type, named appropriately *`type`*`Type`, so that direct comparisons can be performed with results that determine an object's type. In our example, we check to see if `safe_float()` returns a string or float. Any string indicates an error situation with a string that could not be converted to a number, while all other values are floats which can be added to the running subtotal. The final new balance is then displayed as the final line of the `main()` function.

**Lines 38 – 39**

These lines represent the general "start only if not imported" functionality.

Upon running our program, we get the following output:

```
% cardrun.py
$58.94 (new balance)
```

Taking a peek at the resulting log file (`cardlog.txt`), we see that it contains the following log entries after `cardrun.py` processed the transactions found in `carddata.txt`:

```
% cat cardlog.txt
account log:
ignored: invalid literal for float(): # carddata.txt
ignored: invalid literal for float(): previous balance
data… processed
ignored: invalid literal for float(): debits
data… processed
data… processed
data… processed
ignored: invalid literal for float(): credits
data… processed
data… processed
ignored: invalid literal for float(): finance charge/
late fees
data… processed
data… processed
```

## `else` Clause

We have seen the **`else`** statement with other Python constructs such as conditionals and loops. With respect to **`try-except`** statements, its functionality is not that much different from anything else you have seen: The **`else`** clause executes if no exceptions were detected in the preceding **`try`** suite.

All code within the **`try`** suite must have completed successfully (i.e., concluded with no exceptions raised) before any code in the **`else`** suite begins execution. Here is a short example in Python pseudocode:

```
import 3rd_party_module

log = open('logfile.txt', 'w')

try:
   3rd_party_module.function()
except:
```

```
   log.write("*** caught exception in module\n")
else:
   log.write("*** no exceptions caught\n")

log.close()
```

In the above example, we import an external module and test it for errors. A log file is used to determine whether there were defects in the third-party module code. Depending on whether an exception occurred during execution of the external function, we write differing messages to the log.

## `try-except` Kitchen Sink

We can combine all the varying syntaxes that we have seen so far in this chapter to highlight all the different ways you can use `try-except-else:`

```
try:
     try_suite

except
                                                    Exception1:
     suite_for_Exception1

except (Exception2, Exception3, Exception4):
      suite_for_Exceptions_2_3_and_4

except
                                                    Exception5,
Argument5:
     suite_for_Exception5_plus_argument

except (Exception6, Exception7), Argument67:
     suite_for_Exceptions6_and_7_plus_argument

except:
     suite_for_all_other_exceptions

else:
     no_exceptions_detected_suite
```

## `try-finally` Statement

The `try-finally` statement differs from its `try-except` brethren in that it is not used to handle exceptions. Instead it is used to maintain consistent behavior regardless of whether or not exceptions occur. The `finally` suite executes regardless of an exception being triggered within the `try` suite.

```
try:
    try_suite
finally:
    finally_suite  # executes regardless of
exceptions
```

When an exception does occur within the **try** suite, execution jumps immediately to the **finally** suite. When all the code in the **finally** suite completes, the exception is re-raised for handling at the next higher layer. Thus it is common to see a **try-finally** nested as part of a **try-except** suite.

One place where we can add a **try-finally** statement is by improving our code in `cardrun.py` so that we catch any problems which may arise from reading the data from the `carddata.txt` file. In the current code in Example 10.1, we do not detect errors during the read phase (using `readlines()`):

```
try:
     ccfile = open('carddata.txt')
except IOError:
     log.write('no txns this month\n')
     log.close()
     return

txns = ccfile.readlines()
ccfile.close()
```

It is possible for `readlines()` to fail for any number of reasons, one of which is if `carddata.txt` was a file on the network (or a floppy) that became inaccessible. Regardless, we should improve this piece of code so that the entire input of data is enclosed in the **try** clause:

```
try:
    ccfile = open('carddata.txt')
    txns = ccfile.readlines()
    ccfile.close()

except IOError:
    log.write('no txns this month\n')
    log.close()
    return
```

All we did was to move the `readlines()` and `close()` method calls to the try suite. Although our code is more robust now, there is still room for improvement. Notice what happens if there was an error of some sort. If the open succeeds but for some reason the

`readlines()` call does not, the exception will continue with the **`except`** clause. No attempt is made to close the file. Wouldn't it be nice if we closed the file regardless of whether an error occurred or not? We can make it a reality using **`try-finally:`**

```
try:
    ccfile = open('carddata.txt')
    try:
        txns = ccfile.readlines()
    finally:
        ccfile.close()
except IOError:
        log.write('no txns this month\n')
        log.close()
        return
```

Now our code is more robust than ever. Let us take a look at another familiar example, calling `float()` with an invalid value. We will use **`print`** statements to show you the flow of execution within the **`try-except`** and **`try-finally`** clauses. We present `tryfin.py` in Example 10.2.

**Example I0.2. Testing the `try-finally` Statement (`tryfin.py`)**

This small script simply illustrates the flow of control when using a **`try-finally`** statement embedded within the try clause of a **`try-except`** statement.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  try: <$nopage>
004 4       print 'entering 1st try'
005 5       try: <$nopage>
006 6           print 'entering 2nd try'
007 7           float('abc')
008 8
009 9       finally: <$nopage>
010 10          print 'doing finally'
011 11
012 12  except ValueError:
013 13      print 'handling ValueError'
014 14
015 15  print 'finishing execution'
016  <$nopage>
```

Running this code, we get the following output:

```
% tryfin.py
entering 1st try
entering 2nd try
doing finallyhandling ValueError
```

```
finishing execution
```

One final note: If the code in the **finally** suite raises another exception, or is aborted due to a **return,break,** or **continue** statement, the original exception is lost and cannot be re-raised. Quick review: The **try-finally** statement presents a way to detect errors but ignore other than cleanup, and passes the exception up to higher layers for possible handling.

**NOTE**

Currently, ***continue*** statements inside a ***try*** suite are not allowed due to the current implementation of the Python bytecode generator (see FAQ 6.28). This restriction has been lifted in JPython, however.

The proper workaround is to use an ***if-else*** in place of a ***continue.*** A more interesting solution involves creating a special exception handler to issue the ***continue*** (since ***continue*** statements are fine inside an ***except*** clause), as illustrated by the following Python pseudocode:

```
# create our own exception (see section 10.9)
class Continue(Exception):
    pass

# begin our loop
some_loop:              pseudocode for a loop

    # try clause inside some_loop
                                                        try:
                                                        if


    skip_rest_of_loop_expr:
                                                        raise
Continue

         …code we do not want executed
         if skip_rest_of_loop_expr is true…

    except Continue:    # continue proxy (as except clause)
        continue        # start next some_loop iteration

    except

    SomeError:   # handle real exceptions
         :
```

We will look at the ***raise*** statement later on in this chapter, but, as you can probably tell, ***raise*** is the statement that lets programmers explicitly raise exceptions in Python.

## *Exceptions as Strings

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have any relationships to each other. With the advent of exception classes, this is no longer the case. As of 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on.

For backwards compatibility, it is possible to revert to string-based exceptions. Starting the Python interpreter with the command-line option -X will provide you with the standard exceptions as strings. This feature will be obsoleted beginning with Python 1.6.

If you must use string exceptions, we will now show you how to do it right. The following piece of code may or may not work:

```
# this may not work… risky!
try:

          :
     raise 'myexception'
          :
except 'myexception'
     suite_to_handle_my_string_exception
except:
     suite_for_other_exceptions
```

The reason why the above code may not work is because exceptions are based on object identity as opposed to object value (see Section 10.5.1). There are two different string objects above, both with the same value. To rectify the potential problem, create a static string object with which to use:

```
# this is a little bit better
myexception = 'myexception'
try:

          :
     raise myexception
          :
except myexception:
     suite_to_handle_my_string_exception
except:
     suite_for_other_exceptions
```

With this update, the same string object is used. However, if you are going to use this code, you might as well use an exception class. Substitute the `myexception` assignment above with:

```
# this is the best choice
class MyException(Exception):
      pass
            :
try:
            :

     raise MyException
            :

except MyException:
     suite_to_handle_my_string_exception
except:
     suite_for_other_exceptions
```

So you see, there really is no reason *not* to use exception classes from now on when creating your own exceptions. Be careful, however, because you may end up using an external module which may still have exceptions implemented as strings.

## *Exceptions as Classes

As we mentioned above, as of Python 1.5, all standard exceptions are now identified using classes. User-defined, class-based exceptions have been around for longer than that (since Python 1.2!), but until 1.5, the standard exceptions remained implemented as strings, mostly for backwards compatibility. However, there are a number of advantages that classes bring to the table, and these reasons were what finally led to all standard exceptions being converted from strings to class-based.

### Selection via Object Identity

The search for an exception handler (checking each **`except`** clause) is accomplished via object identity and not object value. That means that if you are using string exceptions, the string object used in the **`except`** clause must be the same as the string exception that is raised. Two different string objects, even if they contain exactly the same string, constitute different exceptions!

Using classes simplifies this selection mechanism because exception classes are, for the most part, static. When referring to an exception, you are really accessing a class object that is a built-in identifier and stays constant throughout the course of execution. Whether using `IndexError` in an **`except`** clause or in a **`raise`** statement, you can be sure that they are both referencing the same class object so that a corresponding handler will be found.

### Relationship Between Exceptions

Utilizing classes also allows for a hierarchical structure of exceptions. There are two consequences of employing this construct:

### Promotes Grouping of Related Exceptions

When errors were simply strings, there was no interrelationship between any pair of errors. Although most errors are unrelated, some *are* very closely related, such as `IndexError`–offset into a sequence with an invalid index, and `KeyError`–indexing into a map with an invalid key. String exceptions allow these exceptions to be related in context or description only and do not recognize any more than that codewise.

Class-based exceptions allow such a relationship. Both exceptions now are subclassed from a common ancestor, the `LookupError` exception. If your application defined a new class with a lookup-related error, it is now possible for you to create yet another related exception simply by also subclassing from `LookupError`, or even `IndexError` or `KeyError`.

The complete set of Python exceptions and class hierarchy can be found in Table 10.2

### Simplifies Detection

With class-based exceptions, handler code can detect an entire exception class "tree" (i.e., an ancestor exception class as well as all derived subclasses). As an example, let us say that you just want to catch any general arithmetic error in your program. Our code may be structured something like the following:

```
try:
     code_to_scan_for_math_errors
except FloatingPointError:
     print "math exception found"
except ZeroDivisionError:
     print "math exception found"
except OverflowError:
     print "math exception found"
```

Since the handlers for each exception are the same, we can shorten the code to:

```
try:
     code_to_scan_for_math_errors
except (FloatingPointError, ZeroDivisionError, OverflowError):
    print "math exception found"
```

However, this solution is not as all-encompassing as it could be, is a little messy perhaps with all three exceptions listed, and does not take into account future expansion. What if the next version of Python comes with a new arithmetic exception, or perhaps you create such a new exception for your application? The code we have above would be out-of-date and inaccurate.

The solution is to reference a base class in your `except` clause. Because your new exceptions (as well as `FloatingPointError`, `ZeroDivisionError`, and `OverflowError`) are all subclassed from the `ArithmeticError` exception class, you can reference `ArithmeticError` which can then scan for all `ArithmeticError` exceptions as well as all exceptions *derived* from `ArithmeticError`. Updating our code one more time, we present the most flexible solution here:

```
try:
     code_to_scan_for_math_errors
except ArithmeticError:
     print "math exception found"
```

Now your code can handle all pre-existing `ArithmeticError` exceptions as well as any you may create subclassed from `ArithmeticError`. Care must be taken, however, when handling both classes and superclasses with the same `try` statement. Observe both of the following examples:

```
try:
     code_to_scan_for_math_errors
except ArithmeticError:
     print "math exception found"
except ZeroDivisionError:
     print "division by zero error"

try:
     code_to_scan_for_math_errors
except ZeroDivisionError:
     print "division by zero error"
except ArithmeticError:
     print "math exception found"
```

Exception handlers are mutually-exclusive, meaning that once a handler is found for an exception (or a base class), it is handled immediately without searching further. In the first example, a `ZeroDivisionError` will be handled only by the first `except` statement, producing an output of "math exception found." The `except` clause for `ZeroDivisionError` will not be reached.

The second example may prove to be more useful, as a specific arithmetic error (`ZeroDivisionError`) is handled first, leaving the general `ArithmeticError` handler to take care of any other exception derived from `ArithmeticError`.

## Raising Exceptions

The interpreter was responsible for raising all of the exceptions which we have seen so far. These exist as a result of encountering an error during execution. A programmer

writing an API may also wish to throw an exception on erroneous input, for example, so Python provides a mechanism for the programmer to explicitly generate an exception: the **raise** statement.

## raise Statement

The **raise** statement is quite flexible with the arguments which it supports, translating to a large number of different formats supported syntactically. The general syntax for **raise** is:

```
raise [Exception [, args [, traceback]]]
```

The first argument, Exception, is the name of the exception to raise. If present, it must either be a string, class, or instance (more below). Exception must be given if any of the other arguments (arguments or traceback) are present. A list of all Python standard exceptions is given in Table 10.2.

The second expression contains optional *args* (a.k.a. parameters, values) for the exception. This value is either a single object or a tuple of objects. When exceptions are detected, the exception arguments are always returned as a tuple. If *args* is a tuple, then that tuple represents the same set of exception arguments which are given to the handler. If *args* is a single object, then the tuple will consist solely of this one object (i.e., a tuple with one element). In most cases, the single argument consists of a string indicating the cause of the error. When a tuple is given, it usually equates to an error string, an error number, and perhaps an error location, such as a file, etc.

The final argument, *traceback*, is also optional (and rarely used in practice), and, if present, is the traceback object used for the exception—normally a traceback object is newly created when an exception is raised. This third argument is useful if you want to re-raise an exception (perhaps to point to the previous location from the current). Arguments which are absent are represented by the value **None.**

The most common syntax used is when *Exception* is a class. No additional parameters are ever required, but in this case, if they are given, can be a single object argument, a tuple of arguments, or an exception class instance. If the argument is an instance, then it can be an instance of the given class or a derived class (subclassed from a pre-existing exception class). No additional arguments (i.e., exception arguments) are permitted if the argument is an instance.

What happens if the argument is an instance? No problems arise if instance is an instance of the given exception class. However, if instance is *not* an instance of the class nor an instance of a subclass of the class, then a new instance of the exception class will be created with exception arguments copied from the given instance. If instance is

an instance of a subclass of the exception class, then the new exception will be instantiated from the subclass, not the original exception class.

If the additional parameter to the **raise** statement used with an exception class is not an instance—instead, it is a singleton or tuple—then the class is instantiated and *args* is used as the argument list to the exception. If the second parameter is not present or None, then the argument list is empty.

If *Exception* is an instance, then we do not need to instantiate anything. In this case, additional parameters must not be given or must be None. The exception type is the class which *instance* belongs to; in other words, this is equivalent to raising the class with this instance, i.e., **raise** *instance.__class__, instance.*

Use of string exceptions is deprecated in favor of exception classes, but if *Exception* is a string, then it raises the exception identified by *string,* with any optional parameters (*args*) as arguments.

Finally, the **raise** statement by itself without any parameters is a new construct, introduced in Python 1.5, and causes the last exception raised in the current code block to be re-raised. If no exception was previously raised, a TypeError exception will occur, because there was no previous exception to re-raise.

Due to the many different valid syntax formats for **raise** (i.e., *Exception* can be either a class, instance, or a string), we provide Table 10.1 to illuminate all the different ways which **raise** can be used.

**Table 10.1. Using the raise Statement**

| **raise** *syntax* | *Description* |
|---|---|
| **raise** *exclass* | raise an exception, creating an instance of *exclass* (without any exception arguments) |
| **raise** *exclass()* | same as above since classes are now exceptions; invoking the class name with the function call operator instantiates an instance of *exclass,* also with no arguments |
| **raise** *exclass, args* | same as above, but also providing exception arguments *args,* which can be a single argument or a tuple |
| **raise** *exclass(args)* | same as above |
| **raise** *exclass, args, tb* | same as above, but provides traceback object *tb* to use |
| **raise** *exclass, instance* | raise exception using *instance* (normally an instance of *exclass*); if *instance* is an instance of a subclass of *exclass,* then the new exception will be of the subclass type (not of *exclass* type); if *instance* is *not* an instance of *exclass nor* an instance of a subclass of *exclass,* then a new instance of *exclass* will be created with exception arguments copied from *instance* |
| **raise** instance | raise exception using *instance:* the exception type is the class which instantiated *instance;* equivalent to **raise** |

| | |
|---|---|
| | *instance*.__class__, *instance* (same as above) |
| **raise** *string* | (*archaic*) raises *string* exception |
| **raise** *string*, *args* | same as above, but raises exception with *args* |
| **raise** *string*, *args*, *tb* | same as above, but provides traceback object *tb* to use |
| **raise** | (*new in 1.5*) re-raises previously raised exception; if no exception was previously raised, a TypeError is raised |

## Assertions

Assertions are diagnostic predicates which must evaluate to Boolean true; otherwise, an exception is raised to indicate that the expression is false. These work similarly to the assert macros which are part of the C language preprocessor, but in Python these are run-time constructs (as opposed to pre-compile directives).

If you are new to the concept of assertions, no problem. The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a **raise-if-not** statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the **assert** statement, the newest keyword to Python, introduced in version 1.5.

### **assert** Statement

The **assert** statement evaluates a Python expression, taking no action if the assertion succeeds (similar to a **pass** statement), but otherwise raises an AssertionError exception. The syntax for **assert** is:

**assert** *expression*[, *arguments*]

Here are some examples of the use of the **assert** statement:

```
assert 1 == 1
assert (2 + 2) == (2 * 2)
assert len(['my list', 12]) < 10
assert range(3) == [0, 1, 2]
```

AssertionError exceptions can be caught and handled like any other exception using the **try-except** statement, but if not handled, they will terminate the program and produce a traceback similar to the following:

```
>>> assert 1 == 0
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AssertionError
```

Like the **raise** statement we investigated in the previous section, we can provide an exception argument to our **assert** command:

```
>>> assert 1 == 0, 'One does not equal zero silly!'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AssertionError: One does not equal zero silly!
```

Here is how we would use a **try-except** statement to catch an `AssertionError` exception:

```
try:
    assert 1 == 0, 'One does not equal zero silly!'
except AssertionError, args:
    print '%s: %s' % (args.__class__.__name__, args)
```

Executing the above code from the command-line would result in the following output:

```
AssertionError: One does not equal zero silly!
```

To give you a better idea of how **assert** works, imagine how the **assert** statement may be implemented in Python if written as a function. It would probably look something like this:

```
def assert(expr, args=None):
    if __debug__ and not expr:
        raise AssertionError, args
```

The first **if** statement confirms the appropriate syntax for the assert, meaning that `expr` should be an expression. We compare the type of `expr` to a real expression to verify. The second part of the function evaluates the expression and raises `AssertionError,` if necessary. The built-in variable `__debug__` is 1 under normal circumstances, 0 when optimization is requested (command line option -O).

## Standard Exceptions

Table 10.2 lists all of Python's current set of standard exceptions. All exceptions are loaded into the interpreter as a built-in so they are ready before your script starts or by the time you receive the interpreter prompt, if running interactively.

All standard/built-in exceptions are derived from the root class `Exception`. There are currently two immediate subclasses of `Exception`: `SystemExit` and `StandardError`. All other built-in exceptions are subclasses of `StandardError`. Every level of indentation of an exception listed in Table 10.2 indicates one level of exception class derivation.

**Table 10.2. Python Standard Exceptions**

| *Exception Name* | *Description* |
|---|---|
| `Exception`[a] | root class for all exceptions |
| `SystemExit` | request termination of Python interpreter |
| `StandardError`[a] | base class for all standard built-in exceptions |
| `ArithmeticError`[a] | base class for all numeric calculation errors |
| `FloatingPointError`[a] | error in floating point calculation |
| `OverflowError` | calculation exceeded maximum limit for numerical type |
| `ZeroDivisionError` | division (or modulus) by zero error (all numeric types) |
| `AssertionError`[a] | failure of **`assert`** statement |
| `AttributeError` | no such object attribute |
| `EOFError` | end-of-file marker reached without input from built-in |
| `EnvironmentError`[b] | base class for operating system environment errors |
| `IOError` | failure of input/output operation |
| `OSError`[b] | operating system error |
| `WindowsError`[c] | MS Windows system call failure |
| `ImportError` | failure to import module or object |
| `KeyboardInterrupt` | user interrupted execution (usually by typing ^C) |
| `LookupError`[a] | base class for invalid data lookup errors |
| `IndexError` | no such index in sequence |
| `KeyError` | no such key in mapping |
| `MemoryError` | out-of-memory error (non-fatal to Python interpreter) |
| `NameError` | undeclared/uninitialized object (non-attribute) |
| `UnboundLocalError`[c] | access of an uninitialized local variable |
| `RuntimeError` | generic default error during execution |
| `NotImplementedError`[b] | unimplemented method |
| `SyntaxError` | error in Python syntax |
| `IndentationError`[d] | improper indentation |
| `TableError`[d] | improper mixture of TABs and spaces |
| `SystemError` | generic interpreter system error |
| `TypeError` | invalid operation for type |

| | |
|---|---|
| ValueError | invalid argument given |
| UnicodeError[c] | Unicode related error |

[a] Prior to Python 1.5, the exceptions denoted did not exist. All earlier exceptions were string-based.

[b] New as of Python 1.5.2

[c] New as of Python 1.6

[d] New as of Python 2.0

## *Creating Exceptions

Although the set of standard exceptions is fairly wide-ranging, it may be advantageous to create your own exceptions. One situation is where you would like additional information from what a standard or module-specific exception provides. We will present two examples, both related to `IOError`.

`IOError` is a generic exception used for input/output problems which may arise from invalid file access or other forms of communication. Suppose we wanted to be more specific in terms of identifying the source of the problem. For example, for file errors, we want to have a `FileError` exception which behaves like `IOError`, but with a name that has more meaning when performing file operations.

Another exception we will look at is related to network programming with sockets. The exception generated by the `socket` module is called `socket.error` and is not a built-in exception. It is subclassed from the generic `Exception` exception. However, the exception arguments from `socket.error` closely resemble those of `IOError` exceptions, so we are going to define a new exception called `NetworkError` which subclasses from `IOError` but contains at least the information provided by `socket.error`.

Like classes and object-oriented programming, we have not formally covered network programming at this stage, but skip ahead to Chapter 16 if you need to.

We now present a module called myexc.py with our newly-customized exceptions `FileError` and `NetworkError`. The code is in Example 10.3

### Example 10.3. Creating Exceptions (myexc.py)

This module defines two new exceptions, `FileError` and `NetworkError`, as well as reimplements more diagnostic versions of `open()` [`myopen()`]and `socket.connect()` [`myconnect()`]. Also included is a test function [ `test()`] that is run if this module is executed directly.

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   import  os, socket, errno, types, tempfile
```

```
004 4
005 5   class NetworkError(IOError):
006 6        pass <$nopage>
007 7
008 8   class FileError(IOError):
009 9        pass <$nopage>
010 10
011 11  def updArgs(args, newarg=None):
012 12
013 13      if type(args) == types.InstanceType:
014 14          myargs = []
015 15          for eachArg in args:
016 16              myargs.append(eachArg)
017 17      else: <$nopage>
018 18          myargs = list(args)
019 19
020 20      if newarg:
021 21          myargs.append(newarg)
022 22
023 23      return tuple(myargs)
024 24
025 25  def fileArgs(file, mode, args):
026 26
027 27     if args[0] == errno.EACCES and \
028 28             'access' in dir(os):
029 29         perms = ''
030 30         permd = { 'r': os.R_OK, 'w': os.W_OK,
031 31             'x': os.X_OK}
032 32         pkeys = permd.keys()
033 33         pkeys.sort()
034 34         pkeys.reverse()
035 35
036 36         for eachPerm in 'rwx':
037 37             if os.access(file, permd[eachPerm]):
038 38                 perms = perms + eachPerm
039 39             else: <$nopage>
040 40                 perms = perms + '-'
041 41
042 42         if type(args) == types.InstanceType:
043 43             myargs = []
044 44             for eachArg in args:
045 45                 myargs.append(eachArg)
046 46         else: <$nopage>
047 47             myargs = list(args)
048 48
049 49         myargs[1] = "'%s' %s (perms: '%s')" % \
050 50             (mode, myargs[1], perms)
051 51
052 52         myargs.append(args.filename)
053 53
054 54     else: <$nopage>
055 55         myargs = args
056 56
057 57     return tuple(myargs)
058 58
059 59 def myconnect(sock, host, port):
060 60
```

```
061 61     try: <$nopage>
062 62         sock.connect((host, port))
063 63
064 64     except socket.error, args:
065 65         myargs = updArgs(args)# conv inst2tuple
066 66         if len(myargs) == 1:# no #s on some errs
067 67             myargs = (errno.ENXIO, myargs[0])
068 68
069 69         raise NetworkError, \
070 70             updArgs(myargs, host + ':' + str(port))
071 71
072 72 def myopen(file, mode='r'):
073 73
074 74     try: <$nopage>
075 75         fo = open(file, mode)
076 76
077 77     except IOError, args:
078 78        raise FileError, fileArgs(file, mode, args)
079 79
080 80     return fo
081 81
082 82  def testfile():
083 83
084 84      file = mktemp()
085 85      f = open(file, 'w')
086 86      f.close()
087 87
088 88      for eachTest in ((0, 'r'), (0100, 'r'), \
089 89              0400, 'w'), (0500, 'w')):
090 90          try: <$nopage>
091 91              os.chmod(file, eachTest[0])
092 92              f = myopen(file, eachTest[1])
093 93
094 94          except FileError, args:
095 95              print "%s: %s" % \
096 96                  (args.__class__.__name__, args)
097 97          else: <$nopage>
098 98              print file, "opened ok… perm ignored"
099 99              f.close()
100 100
101 101     os.chmod(file, 0777)# enable all perms
102 102     os.unlink(file)
103 103
104 104 def testnet():
105 105     s = socket.socket(socket.AF_INET, \
106 106         socket.SOCK_STREAM)
107 107
108 108     for eachHost in ('deli', 'www'):
109 109          try: <$nopage>
110 110             myconnect(s, 'deli', 8080)
111 111          except NetworkError, args:
112 112             print "%s: %s" % \
113 113                  (args.__class__.__name__, args)
114 114
115 115 if __name__ == '__main__':
116 116     testfile()
117 117     testnet()
```

```
118  <$nopage>
```

## Lines 1 – 3

The Unix start-up script and importation of the `socket, os, errno, types,` and `tempfile` modules help us start this module.

## Lines 5 – 9

Believe it or not, these five lines make up our new exceptions. Not just one, but both of them. Unless new functionality is going to be introduced, creating a new exception is just a matter of subclassing from an already-existing exception. In our case, that would be `IOError. EnvironmentError,` from which `IOError` is derived would also work, but we wanted to convey that our exceptions were definitely I/O-related.

We chose `IOError` because it provides two arguments, an error number and an error string. File-related [uses `open()`] `IOError` exceptions even support a third argument which is not part of the main set of exception arguments, and that would be the file name. Special handling is done for this third argument which lives outside the main tuple pair and has the name `filename.`

## Lines 11 – 23

The entire purpose of the `updArgs()` function is to "update" the exception arguments. What we mean here is that the original exception is going to provide us a set of arguments. We want to take these arguments and make them part of our new exception, perhaps embellishing or adding a third argument (which is not added if nothing is given— `None` is a default argument which we will study in the next chapter). Our goal is to provide the more informative details to the user so that if and when errors occur, the problems can be tracked down as quickly as possible.

## Lines 25 – 57

The `fileArgs()` function is used only by myopen() [see below]. In particular, we are seeking error `EACCES,` which represents "permission denied." We pass all other `IOError` exceptions along without modification (lines 54–55). If you are curious about `ENXIO, EACCES,` and other system error numbers, you can hunt them down by starting at file `/usr/include/sys/errno.h` on a Unix system, or `C:\Msdev\include\Errno.h` if you are using Visual C++ on Windows.

In line 27, we are also checking to make sure that the machine we are using supports the `os.access()` function, which helps you check what kind of file permissions you have for any particular file. We do not proceed unless we receive both a permission error as well as the ability to check what kind of permissions we have. If all checks out, we set up a dictionary to help us build a string indicating the permissions we have on our file.

The Unix file system uses explicit file permissions for the user, group (more than one user can belong to a "group"), and other (any user other than the owner or someone in the same group as the owner) in read, write, and execute ('r', 'w', 'x') order. Windows supports some of these permissions.

Now it is time to build the permission string. If the file has a permission, its corresponding letter shows up in the string, otherwise a dash ( `-` ) appears. For example, a string of `"rw-"` means that you have read and write access to it. If the string reads `"r-x"`, you have only read and execute access; `"---"` means no permission at all.

After the permission string has been constructed, we create a temporary argument list. We then alter the error string to contain the permission string, something which standard `IOError` exception does not provide. "Permission denied" sometimes seems silly if the system does not tell you what permissions you have to correct the problem. The reason, of course, is security. When intruders do not have permission to access something, the last thing you want them to see is what the file permissions are, hence the dilemma. However, our example here is merely an exercise, so we allow for the temporary "breach of security." The point is to verify whether or not the `os.chmod()` functions call affected file permissions the way they are supposed to.

The final thing we do is to add the file name to our argument list and return the set of arguments as a tuple.

## Lines 59 – 70

Our new `myconnect()` function simply wraps the standard socket method `connect()` to provide an `IOError`-type exception if the network connection fails. Unlike the general `socket.error` exception, we also provide the host name and port number as an added value to the programmer.

For those new to network programming, a host name and port number pair are analogous to an area code and telephone number when you are trying to contact someone. In this case, we are trying to contact a program running on the remote host, presumably a server of some sort; therefore, we require the host's name and the port number that the server is listening on.

When a failure occurs, the error number and error string are quite helpful, but it would be even more helpful to have the exact host-port combination as well, since this pair may be dynamically-generated or retrieved from some database or name service. That is the value-add we are bestowing to our version of `connect()`. Another issue arises when a host cannot be found. There is no direct error number given to us by the `socket.error` exception, so to make it conform to the `IOError` protocol of providing an error number-error string pair, we find the closest error number that matches. We choose `ENXIO`.

## Lines 72 – 80

Like its sibling `myconnect(), myopen()` also wraps around an existing piece of code. Here, we have the `open()` function. Our handler catches only `IOError` exceptions. All others will pass through and on up to the next level (when no handler is found for them). Once an **`IOError`** is caught, we raise our own error and customized arguments as returned from `fileArgs().`

## Lines 82 – 102

We shall perform the file testing first, here using the `testfile()` function. In order to begin, we need to create a test file that we can manipulate by changing its permissions to generate permission errors. The `tempfile` module contains code to create temporary file names or temporary files themselves. We just need the name for now and use our new `myopen()` function to create an empty file. Note that if an error occurred here, there would be no handler, and our program would terminate fatally—the test program should not continue if we cannot even *create* a test file.

Our test uses four different permission configurations. A zero means no permissions at all, 0100 means execute-only, 0400 indicates read-only, and 0500 means read- and execute-only (0400 + 0100). In all cases, we will attempt to open a file with an invalid mode. The `os.chmod()` function is responsible for updating a file's permission modes. (NOTE: these permissions all have a leading zero in front, indicating that they are octal [base 8] numbers.)

If an error occurs, we want to display diagnostic information similar to the way the Python interpreter performs the same task when uncaught exceptions occur, and that is giving the exception name followed by its arguments. The `__class__` special variable provides the class object for which an instance was created from. Rather than displaying the entire class name here (`myexc.FileError`), we use the class object's `__name__` variable to just display the class name (`FileError`), which is also what you see from the interpreter in an unhandled error situation. Then the arguments which we arduously put together in our wrapper functions follow.

If the file opened successfully, that means the permissions were ignored for some reason. We indicate this with a diagnostic message and close the file. Once all tests have been completed, we enable all permissions for the file and remove it with the `os.unlink()` function. `os.remove()` is equivalent to `os.unlink().`)

## Lines 104 – 113

The next section of code (`testnet()`) tests our `NetworkError` exception. A socket is a communication endpoint with which to establish contact with another host. We create such an object, then use it in an attempt to connect to a host with no server to accept our connect request and a host not on our network.

## Lines 115 – 117

We want to execute our `test*()` functions only when invoking this script directly, and that is what the code here does. Most of the scripts given in this text utilize the same format.

Running this on a Unix machine, we get the following output:

```
% myexc.py
FileError: [Errno 13] 'r' Permission denied (perms: '---'):
 '/usr/tmp/@18908.1'
FileError: [Errno 13] 'r' Permission denied (perms: '--x'):
 '/usr/tmp/@18908.1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r--'):
 '/usr/tmp/@18908.1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'):
 '/usr/tmp/@18908.1'
NetworkError: [Errno 146] Connection refused: 'deli:8080'
NetworkError: [Errno 6] host not found: 'www:8080'
```

The results are slightly different on a Windows machine:

```
D:\python>python myexc.py
C:\WINDOWS\TEMP\~-195619-1 opened ok… perms ignored
C:\WINDOWS\TEMP\~-195619-1 opened ok… perms ignored
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'):
 'C:\\WINDOWS\\TEMP\\~-195619-1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'):
 'C:\\WINDOWS\\TEMP\\~-195619-1'
NetworkError: [Errno 10061] winsock error: 'deli:8080'
NetworkError: [Errno 6] host not found: 'www:8080'
```

You will notice that Windows does not support read permissions on files, which is the reason why the first two file open attempts succeeded. Your mileage may vary (YMMV) on your own machine and operating system.

## Why Exceptions (Now)?

There is no doubt that errors will be around as long as software is around. The difference in today's fast-paced computing world is that our execution environments have changed, and so has our need to adapt error-handling to accurately reflect the operating context of the software which we develop. Modern-day applications generally run as self-contained graphical user interfaces (GUIs) or in a client-server architecture such as the Web.

The ability to handle errors at the application level has become even more important recently in that users are no longer the only ones directly running applications. As the Internet and online electronic commerce become more pervasive, web servers will be the primary users of application software. This means that applications cannot just fail or

crash outright anymore, because if they do, system errors translate to browser errors, and these in turn lead to frustrated users. Losing eyeballs means losing advertising revenue and potentially significant amounts of irrecoverable business.

If errors do occur, they are generally attributed to some invalid user input. The execution environment must be robust enough to handle the application-level error and be able to produce a user-level error message. This must translate to a "non-error" as far as the web server is concerned because the application must complete successfully, even if all it does is return an error message to present to the user as a valid hypertext markup language (HTML) web page displaying the error.

If you are not familiar with what I am talking about, does a plain web browser screen with the big black words saying, "Internal Server Error" sound familiar? How about a fatal error that brings up a pop-up that declares "Document contains no data"? As a user, do either of these phrases mean anything to you? No, of course not (unless you are an Internet software engineer), and to the average user, they are an endless source of confusion and frustration. These errors are a result of a failure in the execution of an application. The application either returns invalid hypertext transfer protocol (HTTP) data or terminates fatally, resulting in the web server throwing its hands up into the air, saying, "I give up!"

This type of faulty execution should not be allowed, if at all possible. As systems become more complex and involve more apprentice users, additional care should be taken to ensure a smooth user application experience. Even in the face of an error situation, an application should terminate successfully, as to not affect its execution environment in a catastrophic way. Python's exception handling promotes mature and correct programming.

## Why Exceptions at All?

If the above section was not motivation enough, imagine what Python programming might be like without program-level exception handling. The first thing that comes to mind is the loss of control client programmers have over their code. For example, if you created an interactive application which allocates and utilizes a large number of resources, if a user hit ^C or other keyboard interrupt, the application would not have the opportunity to perform clean-up, resulting in perhaps loss of data, or data corruption. There is also no mechanism to take alternative action such as prompting the users to confirm whether they really want to quit or if they hit the control key accidentally.

Another drawback would be that functions would have to be rewritten to return a "special" value in the face of an error situation, for example, `None`. The engineer would be responsible for checking each and every return value from a function call. This may be cumbersome because you may have to check return values which may not be of the same type as the object you are expecting if no errors occurred. And what if your function wants to return None as a valid data value? Then you would have to come up with another return value, perhaps a negative number. We probably do not need to remind you that negative numbers may be valid in a Python context, such as an index into a sequence. As a programmer of application programmer interfaces (APIs), you would then have to

document every single return error your users may encounter based on the input received. Also, it is difficult (and tedious) to propagate errors (and reasons) of multiple layers of code.

There is no simple propagation like the way exceptions do it. Because error data needs to be transmitted upwards in the call hierarchy, it is possible to misinterpret the errors along the way. A totally unrelated error may be stated as the cause when in fact it had nothing to do with the original problem to begin with. We lose the bottling-up and safekeeping of the original error that exceptions provide as they are passed from layer to layer, not to mention completely losing track of the data we were originally concerned about! Exceptions simplify not only the code, but the entire error management scheme which should not play such a significant role in application development. And with Python's exception handling capabilities, it does not have to.

## Exceptions and the `sys` Module

An alternative way of obtaining exception information is by accessing the `exc_info()` function in the `sys` module. This function provides a 3-tuple of information, more than what we can achieve by simply using only the exception argument. Let us see what we get using `sys.exc_info()`:

```
>>> try:
…       float('abc123')
… except:
…       import sys
…       exc_tuple = sys.exc_info()
…
>>> print exc_tuple
(<class exceptions.ValueError at f9838>, <exceptions.ValueError
instance at 122fa8>,
<traceback object at 10de18>)
>>>
>>> for eachItem in exc_tuple:
…       print eachItem
…
exceptions.ValueError
invalid literal for float(): abc123
<traceback object at 10de18>
```

What we get from sys.`exc_info()` in a tuple are:

exception class object

(this) exception class instance object

traceback object

The first two items we are familiar with: the actual exception class and this particular exception's instance (which is the same as the exception argument which we discussed in the previous section). The third item, a traceback object, is new. This object provides the execution context of where the exception occurred. It contains information such as the execution frame of the code that was running and the line number where the exception occurred.

In older versions of Python, these three values were available in the `sys` module as `sys.exc_type, sys.exc_value,` and `sys.exc_traceback.` Unfortunately, these three are global variables and not thread-safe. We recommend using `sys.exc_info()` instead.

## Related Modules

The classes found in module `Lib/exceptions.py` are automatically loaded as built-in names on start-up, so no explicit import of this module is ever necessary. We recommend you take a look at this source code to familiarize yourself with Python's exceptions and how they interrelate and interoperate. Starting with 2.0, exceptions are now built into the interpreter (see `Python/exceptions.c`).

## Exercises

**1:**

*Raising Exceptions.* Which of the following can RAISE exceptions during program execution? Note that this question does not ask what may CAUSE exceptions.

a) the user

b) the interpreter

c) the program

d) all of the above

e) only (b) and (c)

f) only (a) and (c)

**2:**

*Raising Exceptions.* Referring to the list in the problem above, which could raise exceptions while running within the interactive interpreter?

**3:**

*Keywords.* Name the keyword(s) which is(are) used to raise exceptions.

**4:**

*Keywords.* What is the difference between **`try-except`** and **`try-finally?`**

**5:**

*Exceptions.* Name the exception that would result from executing the following pieces of Python code from within the interactive interpreter (refer back to Table 10.2 for a list of all built-in exceptions):

a)

```
>>> if 3 < 4 then: print '3 IS less than 4!'
```

b)

```
>>> aList = ['Hello', 'World!', 'Anyone', 'Home?']
>>> print 'the last string in aList is:', aList[len(aList)]
```

c)

```
>>> x
```

d)

```
>>> x = 4 % 0
```

e)

```
>>> import math
>>> i = math.sqrt(-1)
```

**6:**

*Improving `open()`.* Create a wrapper for the `open()` function. When a program opens a file successfully, a file handle will be returned. If the file open fails, rather than generating an error, return `None` to the callers so that they can open files without an exception handler.

**7:**

*Exceptions.* What is the difference between Python pseudocode snippets (a) and (b)? Answer in the context of statements A and B, which are part of both pieces of code. (Thanks to Guido for this teaser!)

a)

```
try:
    statement_A
except …:
       …
else:
    statement_B
```

b)

```
try:
    statement_A
    statement_B
except …:
       …
```

**8:**

*Improving* `raw_input().` In the beginning of this chapter, we presented a "safe" version of the `float()` built-in function to detect and handle two different types of exceptions which `float()` generates. Likewise, the `raw_input()` function can generate two different exceptions, either `EOFError` or `KeyboardInterrupt` on end-of-file (EOF) or cancelled input, respectively. Create a wrapper function, perhaps `safe_input();` rather than raising an exception if the user entered EOF (^D in Unix or ^Z in DOS) or attempted to break out using ^C, have your function return None that the calling function can check for.

**9:**

*Improving* `math.sqrt().` The `math` module contains many functions and some constants for performing various mathematics-related operations. Unfortunately, this module does not recognize or operate on complex numbers, which is the reason why the `cmath` module was developed. Rather than suffering the overhead of importing an entire module for complex numbers which you do not plan on using in your application, you want to just use the standard arithmetic operators which work fine with complex numbers, but really want just a square root function that can provide a complex number result when given a negative argument. Create a function, perhaps `safe_sqrt(),` which wraps `math.sqrt(),` but is smart enough to handle a negative parameter and return a complex number with the correct value back to the caller.

# Chapter 11. Functions

We were introduced to functions in Chapter 2 and have seen them created and called throughout the text. In this chapter, we will look beyond the basics and give you a full treatment of all the other features associated with functions. In addition to the expected behavior, functions in Python support a variety of invocation styles and argument types, including some functional programming interfaces. We conclude this chapter with a look at Python's scoping as well as take an optional side trip into the world of recursion.

## What Are Functions?

Functions are the structured or procedural programming way of organizing the logic in your programs. Large blocks of code can be neatly segregated into manageable chunks, and space is saved by putting oft-repeated code in functions as opposed to multiple copies everywhere—this also helps with consistency because changing the single copy means you do not have to hunt for and make changes to multiple copies of duplicated code. The basics of functions in Python are not much different from those of other languages with which you may be familiar. After a bit of review here in the early part of this chapter, we will focus on what else Python brings to the table.

Functions can appear in different ways… here is a sampling profile of how you will see functions created, used, or otherwise referenced:

| | |
|---|---|
| ***declaration/definition*** | `def foo(): print 'bar'` |
| ***function object/reference*** | `foo` |
| ***function call/invocation*** | `foo()` |

### Functions vs. Procedures

Functions are often compared to procedures. Both are entities which can be invoked, but the traditional function or "black box," perhaps taking some or no input parameters, performs some amount of processing and concludes by sending back a return value to the caller. Some functions are Boolean in nature, returning a "yes" or "no" answer, or, more appropriately, a non-zero or zero value, respectively. Procedures, often compared to functions, are simply special cases, functions which do not return a value. As you will see below, Python procedures are implied functions because the interpreter implicitly returns a default value of `None`.

### Return Values and Function Types

Functions may return a value back to its caller and those which are more procedural in nature do not explicitly return anything at all. Languages which treat procedures as functions usually have a special type or value name for functions that "return nothing." These functions default to a return type of `"void"` in C, meaning no value returned. In Python, the equivalent return object type is `None.`

The `hello()` function acts as a procedure in the code below, returning no value. If the return value is saved, you will see that its value is `None:`

```
>>> def hello():
…       print 'hello world'
>>>
>>> res = hello()
hello world
>>> res
>>> print res
None
>>> type(res)
<type 'None'>
```

Also, like most other languages, you may return only one value/object from a function in Python. One difference is that in returning a container type, it will seem as if you can actually return more than a single object. In other words, you can't leave the grocery store with multiple items, but you can throw them all in a single shopping bag which you walk out of the store with, perfectly legal.

```
def foo():
    return ['xyz', 1000000, -98.6]

def bar():
    return 'abc', [42, 'python', "Guido"
```

The `foo()` function returns a list, and the `bar()` function returns a tuple. Because of the tuple's syntax of not requiring the enclosing parentheses, it creates the perfect illusion of returning multiple items. If we were to properly enclose the tuple items, the definition of `bar()` would look like:

```
def bar():
    return ('abc', [4-2j, 'python'], "Guido")
```

As far as return values are concerned, tuples can be saved in a number of ways. The following three ways of saving the return values are equivalent:

```
>>> aTuple = bar()
>>> x, y, z = bar()
>>> (a, b, c) = bar()
>>>
>>> aTuple
('abc', [(4-2j), 'python'], 'Guido')
>>> x, y, z
('abc', [(4-2j), 'python'], 'Guido')
>>> (a, b, c)
('abc', [(4-2j), 'python'], 'Guido')
```

In the assignments for `x, y, z,` and `a, b, c,` each variable will receive its corresponding return value in the order the values are returned. The `aTuple` assignment takes the entire implied tuple returned from the function. Recall that a tuple can be "unpacked" into individual variables or not at all and its reference assigned directly to a single variable. (Refer back to Section 6.17.3 for a review.)

Many languages which support functions maintain the notion that a function's type is the type of its return value. In Python, no direct type correlation can be made since Python is dynamically-typed and functions can return values of different types. Because overloading is not a feature, the programmer can use the `type()` built-in function as a proxy for multiple declarations with different *signatures* (multiple prototypes of the same overloaded function which differ based on its arguments).

## Calling Functions

### Function Operator

Functions are called using the same pair of parentheses that you are used to. In fact, some consider ( `( )` ) to be a two-character operator, the function operator. As you are probably aware, any input parameters or arguments must be placed between these calling parentheses. Parentheses are also used as part of function declarations to define those arguments. Although we have yet to formally study classes and object-oriented programming, you will discover that the function operator is also used in Python for class instantiation.

### Keyword Arguments

The concept of keyword arguments applies only to function invocation. The idea here is for the caller to identify the arguments by parameter name in a function call. This specification allows for arguments to be missing or out-of-order because the interpreter is able to use the provided keywords to match values to parameters.

For a simple example, imagine a function `foo()` which has the following pseudocode definition:

```
def foo(x):
    foo_suite   # presumably does so processing with 'x'
```

Standard calls to `foo()`: `foo(42)` `foo('bar')` `foo(y)`

Keyword calls to `foo()`: `foo(x=42)` `foo(x='bar')` `foo(x=y)`

For a more realistic example, let us assume you have a function called `net_conn()` and you know that it takes two parameters, say, `host` and `port`:

```
def net_conn(host, port):
    net_conn_suite
```

Naturally, we can call the function giving the proper arguments in the correct positional order which they were declared:

```
net_conn('kappa', 8080)
```

The `host` parameter gets the string `'kappa'` and `port` gets 8080. Keyword arguments allow out-of-order parameters, but you must provide the name of the parameter as a "keyword" to have your arguments match up to their corresponding argument names, as in the following:

```
net_conn(port=8080, host='chino')
```

Keyword arguments may also be used when arguments are allowed to be "missing." These are related to functions which have default arguments, which we will introduce in the next section.

### Default Arguments

Default arguments are those which are declared with default values. Parameters which are not passed on a function call are thus allowed and are assigned the default value. We will cover default arguments more formally in Section 11.5.2

## Creating Functions

### `def` Statement

Functions are created using the **def** statement, with a syntax like the following:

```
def
        function_name(arguments):
    "function_documentation_string"
    function_body_suite
```

The header line consists of the **def** keyword, the function name, and a set of arguments (if any). The remainder of the **def** clause consists of an optional but highly-recommended documentation string and the required function body suite. We have seen many function declarations throughout this text, and here is another:

```
def helloSomeone(who):
    'returns a salutory string customized with the input'
    return "Hello" + str(who)
```

## Declaration vs. Definition

Some programming languages differentiate between function declarations and function definitions. A function declaration consists of providing the parser with the function name, and the names (and traditionally the types) of its arguments, without necessarily giving any lines of code for the function, which is usually referred to as the function definition.

In languages where there is a distinction, it is usually because the function definition may belong in a physically different location in the code from the function declaration. Python does not make a distinction between the two, as a function clause is made up of a declarative header line which is immediately followed by its defining suite.

## Forward References

Like some other high-level languages, Python does not permit you to reference or call a function before it has been declared. We can try a few examples to illustrate this:

```
def foo():
    print 'in foo()'
    bar()
```

If we were to call `foo()` here, it will fail because `bar()` has not been declared yet:

```
>>> foo()
in foo()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in foo
NameError: bar
```

We will now define `bar()`, placing its declaration before `foo()`'s declaration:

```
def bar():
    print 'in bar()'

def foo():
    print 'in foo()'
    bar()
```

Now we can safely call `foo()` with no problems:

```
>>> foo()
in foo()
in bar()
```

In fact, we can even declare `foo()` before `bar():`

```
def foo():
    print 'in foo()'
    bar()

def bar():
    print 'in bar()'
```

Amazingly enough, this code still works fine with no forward referencing problems:

```
>>> foo()
in foo()
in bar()
```

This piece of code is fine because even though a call to `bar()` (from `foo()`) appears before `bar()`'s definition, `foo()` *itself* is not called before `bar()` is declared. In other

words, we declared `foo()`, then `bar()`, and *then* called `foo()`, but by that time, `bar()` existed already, so the call succeeds.

Notice that the output of `foo()` succeeded before the error came about. `NameError` is the exception that is always raised when any uninitialized identifiers are accessed.

## Passing Functions

The concept of function pointers is an advanced topic when learning a language such as C, but not Python where functions are like any other object. They can be referenced (accessed or aliased to other variables), passed as arguments to functions, be elements of container objects like lists and dictionaries, etc. The one unique characteristic of functions which may set them apart from other objects is that they are callable, i.e., can be invoked via the function operator. (There are other callables in Python. For more information see Chapter 14)

In the description above, we noted that functions can be aliases to other variables. Because all objects are passed by reference, functions are no different. When assigning to another variable, you are assigning the reference to the same object; and if that object is a function, then all aliases to that same object are invokable:

```
>>> def foo():
…       print 'in foo()'
…
>>> bar = foo
>>> bar()
in foo()
```

When we assigned `foo` to `bar`, we are assigning the same function object to `bar`, thus we can invoke `bar()` in the same way we call `foo()`. Be sure you understand the difference between "`foo`" (reference of the function object) and "`foo()`" (invocation of the function object)

Taking our reference example a bit further, we can even pass functions in as arguments to other functions for invocation:

```
>>> def bar(argfunc):
…       argfunc()
…
>>> bar(foo)
in foo()
```

Note that it is the function object `foo` that is being passed to `bar()`.`bar()` is the function that actually calls `foo()` (which has been aliased to the local variable `argfunc` in the

same way that we assigned `foo` to `bar` in the previous example). Now let us examine a more realistic example, `numconv.py`, whose code is given in Example 11.1.

**Example 11.1. Passing and Calling (Built-in) Functions (`numconv.py`)**

A more realistic example of passing functions as arguments and invoking them from within the function. This script simply converts a sequence of numbers to the same type using the conversion function that is passed in. In particular, the `test()` function passes in a built-in function `int()`, `long()`, or `float()` to perform the conversion.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  def convert(func, seq):
004 4      'conv. sequence of numbers to same type'
005 5      newSeq = []
006 6      for eachNum in seq:
007 7          newSeq.append(func(eachNum))
008 8      return newSeq
009 9
010 10 def test():
011 11     'test function for numconv.py'
012 12     myseq = (123, 45.67, -6.2e8, 999999999L)
013 13     print convert(int, myseq)
014 14     print convert(long, myseq)
015 15     print convert(float, myseq)
016 16
017 17 if __name__ == '__main__':
018 18     test()
019  <$nopage>
```

If we were to run this program, we would get the following output:

```
% numconv.py
[123, 45, -620000000, 999999999]
[123L, 45L, -620000000L, 999999999L]
[123.0, 45.67, -620000000.0, 999999999.0]
```

## Formal Arguments

A Python function's set of formal arguments consists of all parameters passed to the function on invocation for which there is an exact correspondence to those of the argument list in the function declaration. These arguments include all required arguments (passed to the function in correct positional order), keyword arguments (passed in- or out-of-order, but which have keywords present to match their values to their proper positions in the argument list), and all arguments which have default values which may or may not be part of the function call. For all of these cases, a name is created for that value in the (newly-created) local namespace and can be accessed as soon as the function begins execution.

## Positional Arguments

These are the standard vanilla parameters that we are all familiar with. Positional arguments must be passed in the exact order that they are defined for the functions that are called. Also, without the presence of any default arguments (see next section), the exact number of arguments passed to a function (call) must be exactly the number declared:

```
>>> def foo(who):       # defined for only 1 argument
…       print 'Hello', who
…
>>> foo()               # 0 arguments… BAD
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: not enough arguments; expected 1, got 0
>>>
>>> foo('World!')       # 1 argument… WORKS
Hello World!
>>>
>>> foo('Mr.', 'World!')# 2 arguments… BAD
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: too many arguments; expected 1, got 2
```

The `foo()` function has one positional argument. That means that any call to `foo()` must have exactly one argument, no more, no less. You will become extremely familiar with `TypeError` otherwise. Note how informative the Python errors are. As a general rule, all positional arguments for a function must be provided whenever you call it. They may be passed into the function call in position or out-of-position, granted that a keyword argument is provided to match it to its proper position in the argument list (review Section 11.2.2). Default arguments, however, do not have to be provided because of their nature.

## Default Arguments

Default arguments are parameters which are defined to have a default value if one is not provided in the function call for that argument. Such definitions are given in the function declaration header line. C++ and Java are other languages which support default arguments and whose declaration syntax is shared with Python: The argument name is followed by an “assignment of its default value. This assignment is merely a syntactical way of indicating that this assignment will occur if no value is passed in for that argument.

The syntax for declaring variables with default values in Python is such that all positional arguments must come before any default arguments:

```
def                                                    function_name(posargs,
defarg1=dval1, defarg2=dval2,…):
    "function_documentation_string"
    function_body_suite
```

Each default argument is followed by an assignment statement of its default value. If no value is given during a function call, then this assignment is realized.

## Why Default Arguments?

Default arguments add a wonderful level of robustness to applications because they allow for some flexibility that is not offered by the standard positional parameters. That gift comes in the form of simplicity for the applications programmer. Life is not as complicated when there are a fewer number of parameters that one needs to worry about. This is especially helpful when one is new to an API interface and does not have enough knowledge to provide more targeted values as arguments.

The concept of using default arguments is analogous to the process of installing software on your computer. How often does one chose the "default install" over the "custom install?" I would say probably almost always. It is a matter of convenience and know-how, not to mention a timesaver. And if you *are* one of those gurus who always chooses the custom install, please keep in mind that you are one of the minority.

Another advantage goes to the developer, who is given more control over the software they create for their consumers. When providing default values, they can selectively choose the "best" default value possible, thereby hoping to give the user some freedom of not having to make that choice. Over time, as the user becomes more familiar with the system or API, they may eventually be able to provide their own parameter values, no longer requiring the use of "training wheels."

Here is one example where a default argument comes in handy and has some usefulness in the growing electronic commerce industry:

```
>>> def taxMe(cost, rate=0.0825):
…       return cost + (cost * rate)
…
>>> taxMe(100)
108.25
>>>
>>> taxMe(100, 0.05)
105.0
```

In the example above, the `taxMe()` function takes the cost of an item and produces a total sale amount with sales tax added. The cost is a required parameter while the tax rate is a

default argument (in our example, 8¼ %). Perhaps you are an online retail merchant, with most of your customers coming from the same state or county as your business. Consumers from locations with different tax rates would like to see their purchase totals with their corresponding sales tax rates. To override the default, all you have to do is provide your argument value, such as the case with `taxMe(100, 0.05)` in the above example. By specifying a `rate` of 5%, you provided an argument as the `rate` parameter, thereby overriding or bypassing its default value of 0.0825.

All required parameters must be placed before any default arguments. Why? Simply because they are mandatory, whereas default arguments are not. Syntactically, it would be impossible for the interpreter to decide which values match which arguments if mixed modes were allowed. A `SyntaxError` is raised if the arguments are not given in the correct order:

```
>>> def taxMe2(rate=0.0825, cost):
…       return cost * (1.0 + rate)
…
SyntaxError: non-default argument follows default argument
```

Let us take a look at keyword arguments again, using our old friend `net_conn()`.

```
def net_conn(host, port):
    net_conn_suite
```

As you will recall, this is where you can provide your arguments out-of-order (positionally) if you name the arguments. With the above declarations, we can make the following (regular) positional or keyword argument calls:

- `net_conn('kappa', 8000)`
- `net_conn(port=8080, host='chino')`

However, if we bring default arguments into the equation, things change, though the above calls are still valid. Let us modify the declaration of `net_conn()` such that the `port` parameter has a default value of 80 and add another argument named `stype` (for server type) with a default value of `'tcp'`:

```
def net_conn(host, port=80, stype='tcp'):
    net_conn_suite
```

We have just expanded the number of ways we can call `net_conn()`. The following are all valid calls to `net_conn()`:

```
• net_conn('phaze', 8000, 'udp')       # no def args used
• net_conn('kappa')                    # both def args used
• net_conn('chino', stype='icmp')      # use port def arg
• net_conn(stype='udp', host='solo')   # use port def arg
• net_conn('deli', 8080)               # use stype def arg
• net_conn(port=81, host='chino')      # use stype def arg
```

What is the one constant we see in all of the above examples? The sole required parameter, `host`. There is no default value for `host`, thus it is expected in all calls to `net_conn()`.

Keyword arguments calling prove useful for being able to provide for out-of-order positional arguments, but, coupled with default arguments, they can also be used to "skip over" missing arguments as well, as evidenced from our examples above.

## Default Function Object Argument Example

We will now present yet another example of where a default argument may prove beneficial. The `grabweb.py` script, given in Example 11.2 is a simple script whose main purpose is to grab a web page from the Internet and temporarily store it to a local file for analysis. This type of application can be used to test the integrity of a website's pages or to monitor the load on a server (by measuring connectability or download speed). The `process()` function can be anything we want, presenting an infinite number of uses. The one we chose for this exercise displays the first and last non-blank lines of the retrieved web page. Although this particular example may not prove too useful in the real world, you can imagine what kinds of applications you can build on top of this code.

### Example 11.2. Grabbing Web Pages (`grabweb.py`)

This script downloads a webpage (defaults to local www server) and displays the first and last non-blank lines of the HTML file. Flexibility is added due to both default arguments of the`download()` function which will allow overriding with different URLs or specification of a different processing function.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from urllib import urlretrieve
004 4  from string import strip
005 5
006 6  def firstnonblank(lines):
007 7      for eachLine in lines:
008 8          if strip(eachLine) == '':
009 9              continue <$nopage>
```

```
010 10          else: <$nopage>
011 11              return eachLine
012 12
013 13  def firstlast(webpage):
014 14      f = open(webpage)
015 15      lines = f.readlines()
016 16      f.close()
017 17      print firstnonblank(lines),
018 18      lines.reverse()
019 19      print firstnonblank(lines),
020 20
021 21  def download(url='http://www', \
022 22          process=firstlast):
023 23      try: <$nopage>
024 24          retval = urlretrieve(url)[0]
025 25      except IOError:
026 26          retval = None
027 27      if retval:          # do some
028             processing
029 28          process(retval)
030 29
031 30  if __name__ == '__main__':
032 31      download()
033  <$nopage>
```

Running this script in our environment gives the following output, although your mileage will definitely vary since you will be viewing a completely different web page altogether.

```
% grabweb.py
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final
//EN">
</HTML>
```

## Variable-length Arguments

There may be situations where your function is required to process an unknown number of arguments. These are called *variable-length argument lists.* Variable-length arguments are not named explicitly in function declarations because the number of arguments is unknown before run-time (and even during execution, the number of arguments may be different on successive calls), an obvious difference from formal arguments (positional and default) which *are* named in function declarations. Python supports variable-length arguments in two ways because function calls provide for both keyword and non-keyword argument types.

## Non-keyword Variable Arguments (Tuple)

When a function is invoked, all formal (required and default) arguments are assigned to their corresponding local variables as given in the function declaration. The remaining non-keyword variable arguments are inserted in order into a tuple for access. Perhaps you are familiar with "varargs" in C (i.e., `va_list, va_arg,` and the ellipsis [ … ]). Python

provides equivalent support—iterating over the tuple elements is the same as using `va_arg` in C. For those who are *not* familiar with C or varargs, they just represent the syntax for accepting a variable (not fixed) number of arguments passed in a function call.

The variable-length argument tuple must follow all positional and default parameters, and the general syntax for functions with tuple or non-keyword variable arguments is as follows:

```
def
        function_name([formal_args,] *vargs_tuple):
"function_documentation_string"
     function_body_suite
```

The asterisk operator ( `*` ) is placed in front of the variable that will hold all remaining arguments once all the formal parameters have been exhausted, if any. The tuple is empty if there are no additional arguments given.

As we saw earlier, a `TypeError` exception is generated whenever an incorrect number of arguments is given in the function invocation. By adding a variable argument list variable at the end, we can handle the situation when more than enough arguments are passed to the function because all the extra (non-keyword) ones will be added to the variable argument tuple. (The extra keyword arguments require a keyword variable argument parameter [see the next section].)

As expected, all formal arguments must precede informal arguments for the same reason that positional arguments must come before keyword arguments.

```
def tupleVarArgs(arg1, arg2='defaultB', *theRest):
    'display regular args and non-keyword variable args'
    print 'formal arg 1:', arg1
    print 'formal arg 2:', arg1
    for eachXtrArg in theRest:
        print 'another arg:', eachXtrArg
```

We will now invoke this function to show how variable argument tuples work:

```
>>> tupleVarArgs('abc')
formal arg 1: abc
formal arg 2: defaultB
>>>
>>> tupleVarArgs(23, 4.56)
formal arg 1: 23
formal arg 2: 4.56
>>>
```

```
>>> tupleVarArgs('abc', 123, 'xyz', 456.789)
formal arg 1: abc
formal arg 2: 123
another arg: xyz
another arg: 456.789
```

## Keyword Variable Arguments (Dictionary)

In the case where we have a variable number or extra set of keyword arguments, these are placed into a dictionary where the "keyworded" argument variable names are the keys, and the arguments are their corresponding values. Why must it be a dictionary? Because a pair of items is given for every argument—the name of the argument and its value—so it is a natural fit to use a dictionary to hold these arguments. Here is the syntax of function definitions which use the variable argument dictionary for extra keyword arguments:

```
def
        function_name([formal_args,][*vargst,] **vargsd):
    function_documentation_string
    function_body_suite
```

To differentiate keyword variable arguments from non-keyword informal arguments, a double asterisk ( `**` ) is used. The `**` is overloaded so it should not be confused with exponentiation. The keyword variable argument dictionary should be the last parameter of the function definition prepended with the `'**'`. We now present an example of how to use such a dictionary:

```
def dictVarArgs(arg1, arg2='defaultB', **theRest):
    'display 2 regular args and keyword variable args'
    print 'formal arg1:', dictVarArgs
    print 'formal arg2:', arg2
    for eachXtrArg in theRest.keys():
        print 'Xtra arg %s: %s' % \
            (eachXtrArg, str(theRest[eachXtrArg]))
```

Executing this code in the interpreter, we get the following output:

```
>>> dictVarArgs(1220, 740.0, c='grail'
formal arg1: 1220
formal arg2: 740.0
Xtra arg c: grail
>>>
>>> dictVarArgs(arg2='tales', c=123, d='poe',
a='mystery')
formal arg1: mystery
```

```
formal arg2: tales
Xtra arg c: 123
Xtra arg d: poe
>>>
>>> dictVarArgs('one', d=10, e='zoo', men=('freud',
'gaudi'))
formal arg1: one
formal arg2: defaultB
Xtra arg men: ('freud', 'gaudi')
Xtra arg d: 10
Xtra arg e: zoo
```

Both keyword and non-keyword variable arguments may be used in the same function as long as the keyword dictionary is last and is preceded by the non-keyword tuple, as in the following example:

```
def newfoo(arg1, arg2, *nkw, **kw):
    display regular args and all variable args'
    print 'arg1 is:', arg1
    print 'arg2 is:', arg2
    for eachNKW in nkw:
        print 'additional non-keyword arg:', eachNKW
    for eachKW in kw.keys():
        print "additional keyword arg '%s': %s" % \
            (eachKW, kw[eachKW])
```

Calling our function within the interpreter, we get the following output:

```
>>> newfoo('wolf', 3, 'projects', freud=90, gamble=96)
arg1 is: wolf
arg2 is: 3
additional non-keyword arg: projects
additional keyword arg 'freud': 90
additional keyword arg 'gamble': 96
```

## Calling Functions with Variable Argument Objects

Python 1.6 introduces the ability to explicitly provide groups of variable arguments, both a non-keyword tuple and/or a keyword dictionary. In each of the above examples of variable arguments, the variable arguments provided for in the invocation include individual arguments (see all the examples in the preceding section). Prior to 1.6, this was the only way to call a function with a variable number of arguments.

Function calls have the added ability to take a tuple with contents that will go straight to the non-keyword variable argument tuple and a dictionary containing key-value pairs to add to the keyword variable argument dictionary. The tuple and dictionary may be joined

in the function call by those variable arguments that were given the original way, listed individually. The general function invocation full syntax for variable arguments supported by Python starting with 1.6 is:

```
function_name(formal_args, *nonKWtuple, **KWdict)
```

In the previous section, we saw that the `'*'` and `'**'` constructs are already accepted for function declarations, but now they are valid for function calls as well!

We will now use our friend `newfoo()` defined in the previous section to test the new calling syntax. Our first call to `newfoo()` will use the old-style method of listing all arguments individually, even the variable arguments which follow all the formal arguments:

```
>>> newfoo(10, 20, 30, 40, foo=50, bar=60)
arg1 is: 10
arg2 is: 20
additional non-keyword arg: 30
additional non-keyword arg: 40
additional keyword arg 'foo': 50
additional keyword arg 'bar': 60
```

We will now make a similar call; however, instead of listing the variable arguments individually, we will put the non-keyword arguments in a tuple and the keyword arguments in a dictionary to make the call:

```
>>> newfoo(2, 4, *(6, 8), **{'foo': 10, 'bar': 12})
arg1 is: 2
arg2 is: 4
additional non-keyword arg: 6
additional non-keyword arg: 8
additional keyword arg 'foo': 10
additional keyword arg 'bar': 12
```

Finally, we will make another call but build our tuple and dictionary outside of the function invocation:

```
>>> aTuple = (6, 7, 8)
>>> aDict = {'z': 9}
>>> newfoo(1, 2, 3, x=4, y=5, *aTuple, **aDict)
arg1 is: 1
arg2 is: 2
additional non-keyword arg: 3
```

```
additional non-keyword arg: 6
additional non-keyword arg: 7
additional non-keyword arg: 8
additional keyword arg 'z': 9
additional keyword arg 'x': 4
additional keyword arg 'y': 5
```

Notice how our tuple and dictionary arguments make only a subset of the final tuple and dictionary received within the function call. The additional non-keyword value `'3'` and keyword pairs for `'x'` and `'y'` were also included in the final argument lists even though they were not part of the `'*'` and `'**'` variable argument parameters.

## Functional Programming

Python is not and will probably not ever claim to be a functional programming language, but it does support a number of valuable functional programming constructs. There are also some which behave like functional programming mechanisms but may not be traditionally considered as such. What Python *does* provide comes in the form of four built-in functions and **`lambda`** expressions.

## Anonymous Functions and `lambda`

Python allows one to create *anonymous functions* using the **`lambda`** keyword. They are "anonymous" because they are not declared in the standard manner, i.e., using the **`def`** statement. (Unless assigned to a local variable, such objects do not create a name in any namespace either.) However, as functions, they may also have arguments. An entire lambda "statement" represents an expression, and the body of the lambda expression must also be given on the same line as the declaration. We now present the syntax for anonymous functions using **`lambda:`**

```
lambda [arg1[, arg2, … argN]]: expression
```

Arguments are optional, and if used, are usually part of the expression as well.

**NOTE**

Calling **`lambda`** with an appropriate expression yields a function object which can be used like any other function. They can be passed to other functions, aliased with additional references, be members of container objects, and as callable objects, be invoked (with any arguments, if necessary). When called, these objects will yield a result equivalent to the same expression if given the same arguments. They are indistinguishable from functions which return the evaluation of an equivalent expression.

Before we look at any examples using **lambda,** we would like to review single-line statements and then show the resemblances to **lambda** expressions.

```
def true():
    return 1
```

The above function takes no arguments and always returns 1. Single line functions in Python may be written on the same line as the header. Given that, we can rewrite our `true()` function so that it looks something like the following.

```
def true(): return 1
```

We will present the named functions in this manner for the duration of this chapter because it helps one visualize their **lambda** equivalents. For our `true()` function, the equivalent **lambda** expression (no arguments, returns 1) is:

```
lambda :1
```

Usage of the named `true()` function is fairly obvious, but not for **lambda.** Do we just use it as is, or do we need to assign somewhere? A **lambda** function by itself serves no purpose, as we see here:

```
>>> lambda :1
<function <lambda> at f09ba0>
```

In the above example, we simply created a **lambda** function, but did not save it anywhere nor did we call it. The reference count for this function object is set to 1 on creation of the function object, but because no reference is saved, goes back down to zero and garbage-collected. To keep the object around, we can save it into a variable and invoke it any time after. Perhaps now is a good opportunity:

```
>>> true = lambda :1
>>> true()
1
```

Assigning it looks much more useful here. Likewise, we can assign **lambda** expressions to a data structure such as a list or tuple where, based on some input criteria, we can choose which function to execute as well as what the arguments would be. (In the next section, we will show how to use **lambda** expressions with functional programming constructs.)

Let us now design a function that takes two numeric or string arguments and returns the sum for numbers or the concatenated string. We will show the standard function first, followed by its unnamed equivalent.

```
def add(x, y): return x + y?    lambda x, y: x + y
```

Default and variable arguments are permitted as well, as indicated in the following examples:

```
def usuallyAdd2(x, y=2): return x+y ?  lambda x, y=2: x+y
def showAllAsTuple(*z): return z    ?  lambda *z: z
```

Seeing is one thing, so we will now try to make you believe by showing how you can try them in the interpreter:

```
>>> a = lambda x, y=2: x + y
>>> a(3)
5
>>> a(3,5)
8
>>> a(0)
2
>>> a(0,9)
9
>>>
>>> b = lambda *z: z
>>> b(23, 'zyx')
(23, 'zyx')
>>> b(42)
(42,)
```

One final word on **lambda.** Although it appears that **lambda**'s are a one-line version of a function, they are not equivalent to "inline" statements in C++, whose purpose is bypassing function stack allocation during invocation for performance reasons. A **lambda** expression works just like a function, creating a frame object when called.

## Built-in Functions: apply(), filter(), map(), reduce()

In this section, we will look at the `apply()`, `filter()`, `map()`, and `reduce()` built-in functions as well as give some examples to show how they can be used. These functions provide the functional programming features found in Python. A summary of these functions is given in Table 11.1. All take a function object to somehow invoke.

**Table 11.1. Functional Programming Built-in Functions**

| *Built-in Function* | *Description* |
|---|---|
| `apply(func[, nkw][, kw])` | call *func* with optional arguments, *nkw* for non-keyword arguments and *kw* for keyword arguments; the return value is the return value of the function call |
| `filter(func, seq)` | invokes Boolean function *func* iteratively over each element of *seq*; returns a sequence for those elements for which *func* returned true |
| `map(func, seq1[, seq2…])` | applies function *func* to each element of given sequences(s) and provides return values in a list; if *func* is `None`, *func* behaves as the identity function, returning a list consisting of *n*-tuples for sets of elements of each sequence |
| `reduce(func, seq[, init])` | applies binary function *func* to elements of sequence *seq*, taking a pair at a time (previous result and next sequence item), continually applying the current result with the next value to obtain the succeeding result, finally reducing our sequence to a single return value; if initial value *init* given, first compare will be of *init* and first sequence element rather than the first two sequence elements |

As you may imagine, **`lambda`** functions fit nicely into applications using any of these functions because all of them take a function object with which to execute, and **`lambda`** provides a mechanism for creating functions on the fly.

## *`apply()`

The first built-in function we are looking at is `apply()`. The `apply()` function is the most basic of the four and is simply used to pass in a function object along with any parameters. `apply()` will then invoke that function with the given arguments. There is no special magic here; `apply()` works exactly the way you think it does, so the following pair of calls are practically identical:

```
foo(3, 'pyramid')     ?     apply(foo, (3, 'pyramid'))
```

Alternatively, the arguments can be stored in a tuple, and then the function can be called with `apply()`:

```
args = (4, 'eve', 79)
apply(foo, args)
```

Note this is not the same as `foo(args)` which is calling `foo()` with a single argument (a tuple). Rather, using `apply()` means calling `foo()` with three arguments, the elements of the tuple.

If you wanted to call the built-in function `dir()` from the interpreter, you could either execute it directly, or use `apply()`. In this example, they both have the same effect since no arguments are involved.

```
dir()        ?     apply(dir)
```

Below, we perform both function calls in the interpreter to show you they produce identical results:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>>
>>> apply(dir)
['__builtins__', '__doc__', '__name__']
```

You may be wondering… why would I ever need to use `apply()` when I can just make a function call? Is there ever a need to do the following? Not only does it require more typing on my part, but the syntax is more complicated.

`apply()` can be used as an effective tool in certain situations. One scenario where `apply()` comes in handy is when you need to call a function, but its arguments are generated dynamically. Such situations usually involve assembling an argument list. In our math game in Example 11.3 (`matheasy.py`), we generate a two-item argument list to send to the appropriate arithmetic function.

The `matheasy.py` application is basically an arithmetic math quiz game for children where an arithmetic operation is randomly chosen between addition, subtraction, and multiplication. We use the functional equivalents of these operators, `add()`, `sub()`, and `mul()`, all found in the `operator` module. We then generate the list of arguments (two, since these are binary operators/operations). Then random numbers are chosen as the operands. Since we do not want to support negative numbers in this more elementary edition of this application, we sort our list of two numbers in largest-to-smallest order, then call `apply()` with this argument list and the randomly-chosen arithmetic operator to obtain the correct solution to the posed problem. `apply()` makes a good choice for our application for two reasons:

**Example 11.3. Arithmetic Game Using `apply()` (`matheasy.py`)**

Randomly chooses numbers and an arithmetic function, displays the question, and verifies the results. Shows answer after three wrong tries and does not continue until the user enters the correct answer.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2  from string import lower
003 3  from operator import add, sub, mul
004 4  from random import randint, choice
005 5
006 6  ops = { '+': add, '-': sub, '*': mul }
007 7  MAXTRIES = 2
008 8
009 9  def doprob():
010 10     op = choice('+-*')
011 11     nums = [randint(1,10), randint(1,10)]
012 12     nums.sort() ; nums.reverse()
013 13     ans = apply(ops[op], nums)
014 14     pr = '%d %s %d = ' % (nums[0], op, nums[1])
015 15     oops = 0
016 16     while 1:
017 17         try: <$nopage>
018 18             if int(raw_input(pr)) == ans:
019 19                 print 'correct'
020 20                 break <$nopage>
021 21             if oops == MAXTRIES:
022 22                 print 'answer\n%s%d'%(pr,ans)
023 23             else: <$nopage>
024 24                 print 'incorrect… try again'
025 25                 oops = oops + 1
026 26         except (KeyboardInterrupt, \
027 27                 EOFError, ValueError):
028 28             print 'invalid input… try again'
029 29
030 30 def main():
031 31     while 1:
032 32         doprob()
033 33         try: <$nopage>
034 34             opt = lower(raw_input('Again? '))
035 35         except (KeyboardInterrupt, EOFError):
036 36             print ; break <$nopage>
037 37         if opt and opt[0] == 'n':
038 38             break <$nopage>
039 39
040 40 if __name__ == '__main__':
041 41     main()
042  <$nopage>
```

Argument list hand-built

Function randomly chosen

Since we do not know what our arguments are nor do we know what function we will be calling for each math question posed to the user, `apply()` makes for a flexible solution.

## Lines 1 - 4

Our code begins with the usual Unix start-up line, which, we repeat, will be harmlessly ignored on all non-Unix systems. What follows are three **`from-import`** statements which load `string.lower()` for case-insensitive input verification, `random.randint()` for choosing the arithmetic operands, `random.choice()` for picking the arithmetic operation, and all the arithmetic operations we need from the `operator` module.

## Lines 6 - 7

The global variables we use in this application are a set of operations and their corresponding functions, and a value indicating how many times (three: 0, 1, 2) we allow the user to enter an incorrect answer before we reveal the solution. The function dictionary uses the operator's symbol to index into the dictionary, pulling out the appropriate arithmetic function.

## Lines 9 - 28

The `doprob()` function is the core engine of the application. It randomly picks an operation and generates the two operands, sorting them from largest-to-smallest order in order to avoid negative numbers for subtraction problems. It then invokes `apply()` to call the math function with the values, calculating the correct solution. The user is then prompted with the equation and given three opportunities to enter the correct answer.

## Lines 30-41

The main driver of the application is `main()`, called from the top-level if the script is invoked directly. If imported, the importing function either manages the execution by calling `doprob()`, or calls `main()` for program control. `main()` simply calls `doprob()` to engage the user in the main functionality of the script and prompts the user to quit or to try another problem.

Since the values and operators are chosen randomly, each execution of `matheasy.py` should be different. Here is what we got today (oh, and your answers may vary as well!):

```
% matheasy.py
7 - 2 = 5
correct
Try another? ([y]/n)
7 * 6 = 42
correct
Try another? ([y]/n)
7 * 3 = 20
incorrect… try again
7 * 3 = 22
incorrect… try again
7 * 3 = 23
```

```
sorry… the answer is
7 * 3 = 21
7 * 3 = 21
correct
Try another? ([y]/n)
7 - 5 = 2
correct
Try another? ([y]/n) n
```

Another useful application of `apply()` comes in terms of debugging or performance measurement. You are working on functions that need to be fully tested or run through regressions every night, or that need to be timed over many iterations for potential improvements. All you need to do is to create a diagnostic function that sets up the test environment, then calls the function in question. Because this system should be flexible, you want to allow the testee function to be passed in as an argument. So a pair of such functions, `timeit()` and `testit()`, would probably be useful to the software developer today.

We will now present the source code to one such example of a `testit()` function (see Example 11.4). We will leave a `timeit()` function as an exercise for the reader (see Exercise 11.12 at end of chapter).

This module provides an execution test environment for functions. The `testit()` function takes a function and arguments, then invokes that function with the given arguments under the watch of an exception handler. If the function completes successfully, a return value of 1 packaged withthe return value of the function is sent back to the caller. Any failure returns a status of 0 and the cause of the exception. (`Exception` is the root class for all exceptions; review Chapter 10 for details.)

### Example 11.4. Testing Functions (`testit.py`)

`testit()` invokes a given function with its arguments, returning a 1 packaged with the return value of the function on success and 0 with the cause of the exception on failure.

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   def testit(func, *nkwargs, **kwargs):
004 4
005 5       try:
006 6           retval = apply(func, nkwargs, kwargs)
007 7           result = (1, retval)
008 8       except Exception, diag:
009 9           result = (0, str(diag))
010 10      return result
011 11
012 12  def test():
013 13      funcs = (int, long, float)
014 14      vals = (1234, 12.34, '1234', '12.34')
```

```
015 15
016 16      for eachFunc in funcs:
017 17          print '-' * 20
018 18          for eachVal in vals:
019 19              retval = testit(eachFunc, \
020 20                                  eachVal)
021 21              if retval[0]:
022 22                  print '%s(%s) =' % \
023 23      (eachFunc.__name__, 'eachVal'), retval[1]
024 24              else: <$nopage>
025 25                  print '%s(%s) = FAILED:' % \
026 26      (eachFunc.__name__, 'eachVal'), retval[1]
027 27
028 28  if __name__ == '__main__':
029 29      test()
030  <$nopage>
```

The unit tester function `test()` runs a set of numeric conversion functions on an input set of four numbers. There are two failure cases in this test set to confirm such functionality. Here is the output of running the script:

```
% testit.py
--------------------
int(1234) = 1234
int(12.34) = 12
int('1234') = 1234
int('12.34') = FAILED: invalid literal for int(): 12.34
--------------------
long(1234) = 1234L
long(12.34) = 12L
long('1234') = 1234L
long('12.34') = FAILED: invalid literal for long(): 12.34
--------------------
float(1234) = 1234.0
float(12.34) = 12.34
float('1234') = 1234.0
float('12.34') = 12.34
```

## filter()

The second built-in function we examine in this chapter is `filter()`. Imagine going to an orchard and leaving with a bag of apples you picked off the trees. Wouldn't it be nice if you could run the entire bag through a filter to keep just the good ones? That is the main premise of the `filter()` function.

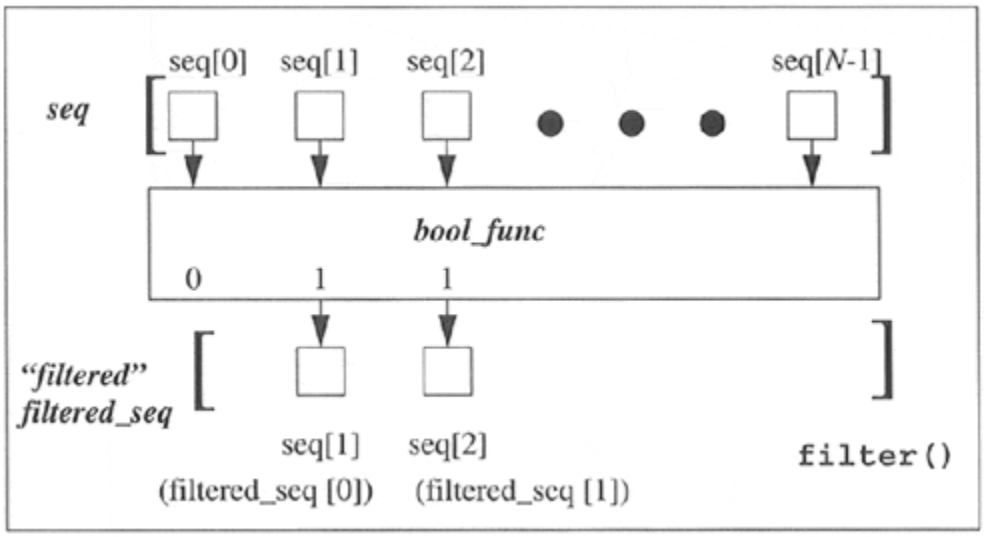
Given a sequence of objects and a "filtering" function, run each item of the sequence through the filter, and keep only the ones that the function returns true for. The `filter()` function calls the given Boolean function for each item of the provided sequence. Each item for which `filter()` returns a non-zero (true) value is appended to a list. The object that is returned is a "filtered" sequence of the original.

If we were to code `filter()` in pure Python, it might look something like this:

```
def filter(bool_func,
                                                    sequence):
filtered_seq = []
for eachItem in
                                                    sequence:
 if apply(bool_func, (eachItem,)):
    filtered_seq.append(eachItem)
    return filtered_seq
```

One way to to understand `filter()` better is by visualizing its behavior. Figure11-1 attempts to do just that.

**Figure 11-1. How the `filter()` Built-in Function Works**




In Figure11-1, we observe our original sequence at the top, items `seq[0]`, `seq[1]`, … `seq[N-1]` for a sequence of size N. For each call to `bool_func()`, i.e., `bool_func(seq[0])`, `bool_func(seq[1])`, etc, a return value of 0 or 1 comes back (as per the definition of a Boolean function—ensure that indeed your function does return 0 or 1). If `bool_func()` returns true for any sequence item, that element is inserted into the return sequence. When iteration over the entire sequence has been completed, `filter()` returns the newly-created sequence.

We present below a script which shows one way to use `filter()` to obtain a short list of random odd numbers. The script generates a larger set of random numbers first, then filters out all the even numbers, leaving us with the desired dataset. When we first coded this example, `oddnogen.py` looked like the following:

```
from random import randint
```

```
def odd(n):
return n % 2

def main():
allNums = []
for eachNum in range(10):
 allNums.append(randint(1, 101))
oddNums = filter(odd, allNums)
print len(oddNums), oddNums

if __name__ == \q__main__\q:
    main()
```

The script consists of two functions: `odd()`, a Boolean function which determined if an integer was odd (true) or even (false), and `main()`, the primary driving component. The purpose of `main()` is to generate ten random numbers between one and a hundred; then `filter()` is called to remove all the even numbers. Finally, the set of odd numbers is displayed, preceded by the size of our filtered list.

Importing and running this module several times, we get the following output:

```
>>> import oddnogen
>>> oddnogen.main()
4 [9, 33, 55, 65]
>>>
>>> oddnogen.main()
5 [39, 77, 39, 71, 1]
>>>
>>> oddnogen.main()
6 [23, 39, 9, 1, 63, 91]
>>>
>>> oddnogen.main()
5 [41, 85, 93, 53, 3]
```

On second glance, we realize that the `odd()` function can be replaced by a **`lambda`** expression to pass to `filter()`, and it is this modification which gives us our final `oddnogen.py` script. The code is given in Example 11.5.

### Example 11.5. Odd Number Generator (`oddnogen.py`)

This simple program generates ten random numbers between one and one hundred, then filters out all the even ones. The program then displays the total number filtered out and the resulting list of odd numbers.

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   from random import randint
```

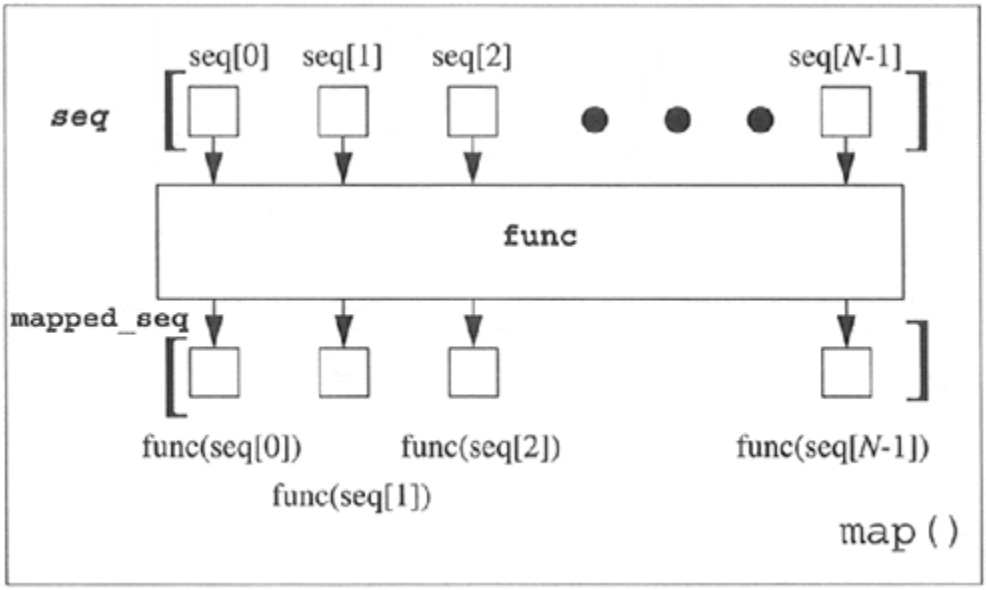
```
004 4
005 5   def main():
006 6
007 7       allNums = []
008 8       for eachNum in range(10):
009 9           allNums.append(randint(1, 100))
010 10      oddNums = filter(lambda n: n % 2, allNums)
011 11      print len(oddNums), oddNums
012 12
013 13  if __name__ == '__main__':
014 14      main()
015  <$nopage>
```

## map()

The `map()` built-in function is similar to `filter()` in that it can process a sequence through a function. However, unlike `filter()`, `map()` "maps" the function call to each sequence item and returns a list consisting of all the return values.

In its simplest form, `map()` takes a function and sequence, applies the function to each item of the sequence, and creates a return value list that is comprised of each application of the function. So if your mapping function is to add 2 to each number that comes in and you feed that function to `map()` along with a list of numbers, the resulting list returned is the same set of numbers as the original, but with 2 added to each number. If we were to code how this simple form of `map()` works in Python, it might look something like the code below and is illustrated in Figure11-2

**Figure 11-2. How the `map()` Built-in Function Works**




```
def map(func, seq):
mapped_seq = []
for eachItem in
```

```
                                                    seq:
 mapped_seq.append(apply(func, eachItem))
    return mapped_seq
```

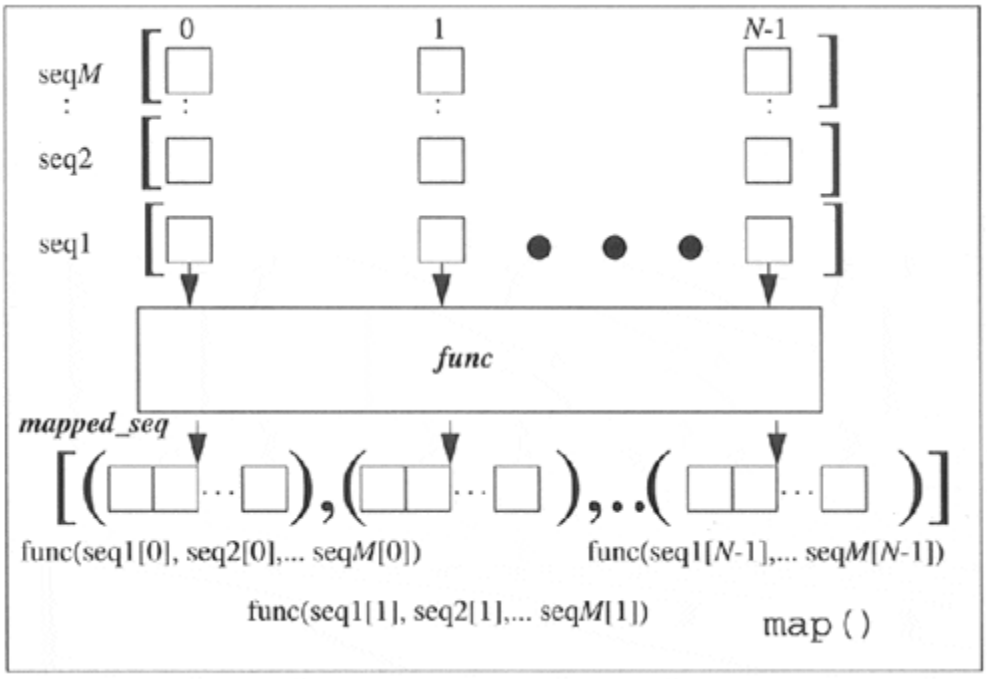
We can whip up a few quick **lambda** functions to show you how map() works on real data:

```
>>> map((lambda x: x+2), [0, 1, 2, 3, 4, 5])
[2, 3, 4, 5, 6, 7]
>>>
>>> map(lambda x: x**2, [0, 1, 2, 3, 4, 5])
[0, 1, 4, 9, 16, 25]
>>>
>>> map((lambda x: x**2), range(6))
[0, 1, 4, 9, 16, 25]
```

The more general form of map() can take more than a single sequence as its input. If this is the case, then map() will iterate through each sequence in parallel. On the first invocation, it will bundle the first element of each sequence into a tuple, apply the func function to it, and return the result as a tuple into the *mapped_seq* mapped sequence that is finally returned as a whole when map() has completed execution.

Figure11-2 illustrated how map() works with a single sequence. If we used map() with *M* sequences of *N* objects each, our previous diagram would be converted to something like the diagram presented in Figure11-3

**Figure 11-3. How the map() Built-in Function Works with > 1 Sequence**




For example, let us consider the following call to `map()`:

```
>>> map(lambda x, y: x + y, [1,3,5], [2,4,6])
[3, 7, 11]
```

In the above example, the number of sequences, *M,* is two. The lists [1, 3, 5] and [2, 4, 6] are our sequences. And each of these sequences has cardinality or size three, or N. The result then consists of the following:

Let:

```
f(x, y) \xba  x + y,
seq0 = [1, 3, 5], and
seq1 = [2, 4, 6]
```

Then:

```
     map(f, seq0, seq1)
=    [ f(seq0[0], seq1[0]), f(seq0[1], seq1[1]), \
         f(seq0[2], seq1[2])
=    [ 1 + 2, 3 + 4, 5 + 6 ]
=    [3, 7, 11]
```

Also, `map()` can also take `None` as the function argument. If `None` is used instead of a real function object, `map()` will take that clue to default to the identity function, meaning that the resulting map will be of the same one as the sequence you passed in. If more than one sequence is passed in, then the resulting list will consist of a tuple with one element from each sequence. Here are a few more examples of using `map()` with multiple sequences, including one with `None` passed in as the `map()` function:

```
>>> map(lambda x, y: (x+y, x-y), [1,3,5], [2,4,6])
[(3, -1), (7, -1), (11, -1)]
>>> map(lambda x, y: (x+y, x*y), [1,3,5], [2,4,6])
[(3, 2), (7, 12), (11, 30)]
>>> map(None, [1,3,5], [2,4,6])
[(1, 2), (3, 4), (5, 6)]
```

This idiom is so commonly used that a new built-in function, `Zip()`, which does the same thing (given sequences of identical size), was added in Python 2.0.

Now these "real-time" examples are nice, but we should also show you some code that you can use in real life. In the next example, we created a text file called `map.txt`, which has a few lines of text surrounded by whitespace. We will use the script `strupper.py`, given in Example 11.6, to strip all the leading and trailing whitespace by passing each line to `string.strip()` and converting all text to uppercase using `string.upper()`. The output of this script will show you the file contents before our manipulation and what the lines look like after we are finished:

### Example 11.6. Text File Processing (`strupper.py`)

`strupper.py`takes an existing text file, strips all leading and trailing whitespace, and converts all the text to uppercase.

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   from string import strip, upper
004 4
005 5   f = open('map.txt')
006 6   lines = f.readlines()
007 7   f.close()
008 8
009 9   print 'BEFORE:\n'
010 10  for eachLine in lines:
011 11      print '[%s]' % eachLine[:-1]
012 12
013 13  print '\nAFTER:\n'
014 14  for eachLine in map(upper, \
015 15          map(strip, lines)):
016 16      print '[%s]' % eachLine
017  <$nopage>
```

```
% strupper.py
BEFORE:

[    Apply function to every item of list and return a ]
[list of the results.  If additional list arguments are    ]
[passed, function must take that many arguments and is     ]
[applied to the  items of all lists in parallel.        ]

AFTER:

[APPLY FUNCTION TO EVERY ITEM OF LIST AND RETURN A]
[LIST OF THE RESULTS.  IF ADDITIONAL LIST ARGUMENTS ARE]
[PASSED, FUNCTION MUST TAKE THAT MANY ARGUMENTS AND IS]
[APPLIED TO THE  ITEMS OF ALL LISTS IN PARALLEL.]
```

Notice that only leading and trailing whitespace is removed. Extra whitespace in the middle of a string such as the last sentence is left as-is.

Our final example in this chapter deals with processing numbers. In particular, assume we have a text file full of numeric dollar amounts. Let us say that these numbers are to go on your income tax form, but you want to round them all to the nearest dollar amount. Here are the contents of our test text file `round.txt`:

```
98.76
90.69
51.36
50.89
28.34
49.64
6.87
36.95
59.25
55.96
```

We now present in Example 11.7 the code to `rounder.py`, a script which strips the trailing NEWLINE character and rounds all the values to the nearest dollar (converting the data from strings to floats first, of course).

**Example 11.7. Text File Number Crunching (`rounder.py`)**

`rounder.py` takes a set of floating point values stored in a text file, and rounds them to the closest whole number. The exercise is to simulate taking numbers destined for income taxes and rounding them to the nearest dollar.

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   f = open('round.txt')
```

```
004 4    values = map(float, f.readlines())
005 5    f.close()
006 6
007 7    print 'original\trounded'
008 8    for eachVal in map(None, values, \
009 9            map(round, values)):
010 10       print '%6.02f\t\t%6.02f' % eachVal
011  <$nopage>
```

The first thing that this script does is to call `map()`, sending each line to the `float()` built-in function, thereby converting the string values to numeric ones while ignoring any leading or trailing whitespace.

Finally, the main part of the code will now present the original values as well as the rounded ones. This is accomplished by sending the values to the `round()` built-in function via `map()`. At the same time, we call `map()` with a `None` function—implying identity which does nothing but merge its sequence arguments into a single list consisting of tuples, each containing one value from each sequence. (Starting in 2.0, we could have also used the zip`()` function, as indicated earlier in this section.) In our case, this constitutes an original value and a rounded value. The **`for`** loop thus iterates over this list of tuples, each tuple representing the original and rounded values, which are then displayed to the user in a nice and readable format.

Executing the `rounder.py` script, we get the following output:

```
% rounder.py
original        rounded
 98.76           99.00
 90.69           91.00
 51.36           51.00
 50.89           51.00
 28.34           28.00
 49.64           50.00
  6.87            7.00
 36.95           37.00
 59.25           59.00
 55.96           56.00
```

### `reduce()`

The final functional programming piece is `reduce()`, which takes a binary function (a function that takes two values, performs some calculation and returns one value as output), a sequence, and an optional initializer, and methodologically "reduces" the contents of that list down to a single value, hence its name.

It does this by taking the first two elements of the sequence and passing them to the binary function to obtain a single value. It then takes this value and the next item of the

sequence to get yet another value, and so on until the sequence is exhausted and one final value is computed.

You may try to visualize `reduce()` as the following equivalence example:

```
reduce(func, [1, 2, 3])      =      func(func(1, 2), 3)
```

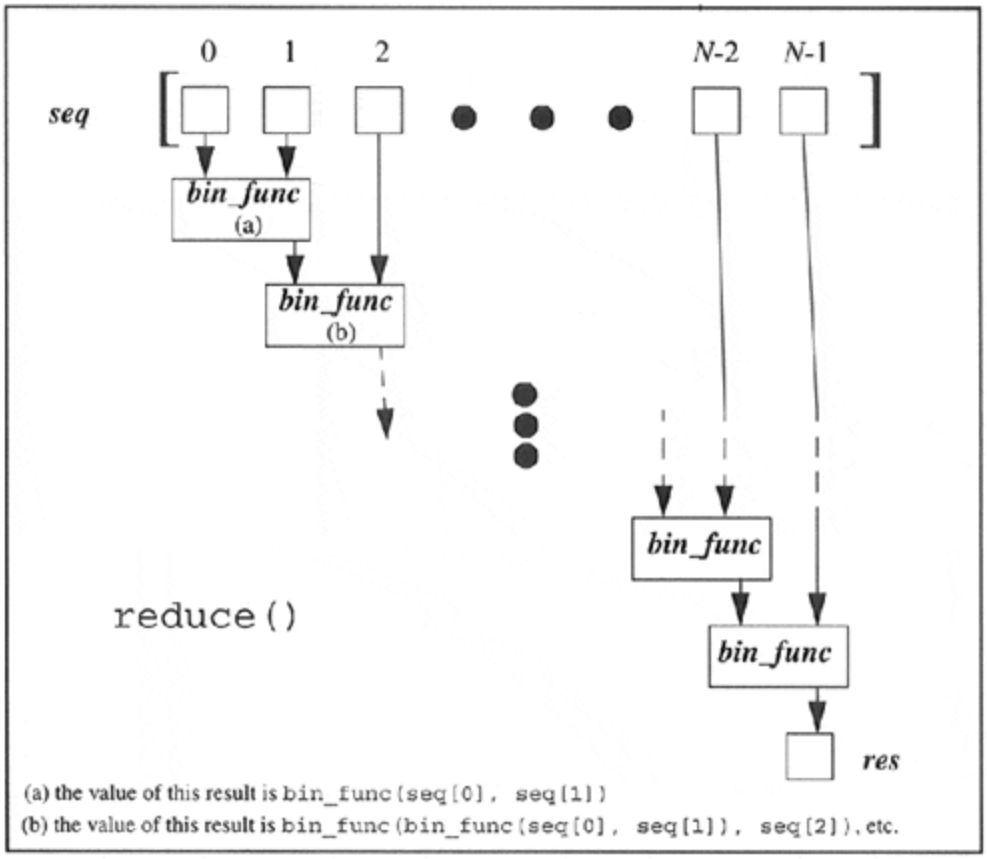
Some argue that the "proper functional" use of `reduce()` requires only one item to be taken at a time for `reduce()`. In our first iteration above, we took two items because we did not have a "result" from the previous values (because we did not *have* any previous values). This is where the optional initializer comes in. If the initializer is given, then the first iteration is performed on the initializer and the first item of the sequence, and follows normally from there.

If we were to try to implement `reduce()` in pure Python, it might look something like this:

```
def reduce(bin_func, seq, init=None):

    lseq = list(seq)                    # convert to list

    if init == None:                    # initializer?
        res = lseq.pop(0)               #   no
    else:
        res = init                      #   yes

    for item in lseq:                  # reduce sequence
        res = bin_func(res, item)       # apply function

    return res                          # return result
```

This may be the most difficult of the four conceptually, so we should again show you an example as well as a functional diagram (see Figure11-4). The "hello world" of `reduce()` is its use of a simple addition function or its `lambda` equivalent seen earlier in this chapter:

**Figure 11-4. How the `reduce()` Built-in Function Works**




- `def sum(x,y): return x,y`
- `lambda x,y: x+y`

Given a list, we can get the sum of all the values by simply creating a loop, iteratively going through the list, adding the current element to a running subtotal, and be presented with the result once the loop has completed:

```
allNums = range(5)                  # [0, 1, 2, 3, 4]
total = 0
for eachNum in allNums:
    total = sum(total, eachNum)     # total = total + eachNum
print 'the total is:', total
```

Making this code real in the interpreter looks like this:

```
>>> def sum(x,y): return x+y
>>> allNums = range(5)
>>> total = 0
>>> for eachNum in allNums:
```

```
...     total = sum(total, eachNum)
...
>>> print 'the total is:' total
the total is: 10
```

Using a `lambda` function, we argue that we can accomplish the same task on a single line using `reduce()`:

```
>>> print 'the total is:', reduce((lambda x,y: x+y), range(5))
the total is: 10
```

The `reduce()` function performs the following mathematical operations given the input above:

```
((0 + 1) + 2) + 3) + 4)?  10
```

It takes the first two elements of the list (0 and 1), calls `sum()` to get 1, then calls `sum()` again with that result and the next item 2, gets the result from that, pairs it with the next item 3 and calls `sum()`, and finally takes the entire subtotal and calls `sum()` with 4 to obtain 10, which is the final return value.

## Variable Scope

The *scope* of an identifier is defined to be the portion of the program where its declaration applies, or what we refer to as "variable visibility." In other words, it is like asking yourself in which parts of a program do you have access to a specific identifier. Variables either have local or global scope.

### Global vs. Local Variables

Variables defined within a function have *local* scope, and those which are at the highest level in a module have *global* or *nonlocal* scope.

In their famous "dragon" book on compiler theory, Aho, Sethi, and Ullman summarize it this way:

"The portion of the program to which a declaration applies is called the *scope* of that declaration. An occurrence of a name in a procedure is said to be *local* to the procedure if it is in the scope of a declaration within the procedure; otherwise, the occurrence is said to be *nonlocal*."

One characteristic of global variables is that unless deleted, they have a lifespan that lasts as long as the script that is running and whose values are accessible to all functions, whereas local variables, like the stack frame they reside in, live temporarily, only as long as the functions they are defined in are currently active. When a function call is made, its local variables come into scope as they are declared. At that time, a new local name is created for that object, and once that function has completed and the frame deallocated, that variable will go out of scope.

```
global_str = 'foo'
def foo():
    local_str = 'bar'
    return global_str + local_str
```

In the above example, `global_str` is a global variable while `local_str` is a local variable. The `foo()` function has access to both global and local variables while the main block of code has access only to global variables.

**NOTE**

When searching for an identifier, Python searches the local scope first. If the name is not found within the local scope, then an identifier must be found in the global scope or else a `NameError` exception is raised.

A variable's scope is related to the namespace in which it resides. We will cover namespaces formally in the next chapter; it suffices to say for now that namespaces are just naming domains which maps names to objects, a virtual set of what variable names are currently in use, if you will. The concept of scope relates to the namespace search order that is used to find a variable. All names in the local namespace are within the local scope when a function is executing. That is the first namespace searched when looking for a variable. If it is not found there, then perhaps a globally-scoped variable with that name can be found. These variables are stored (and searched) in the global and built-in namespaces.

It is possible to "hide" or override a global variable just by creating a local one. Recall that the local namespace is searched first, being in its local scope. If the name is found, the search does not continue to search for a globally-scoped variable, hence overriding any matching name in either the global or built-in namespaces.

Also, be careful when using local variables with the same names as global variables. If you use such names in a function (to access the global value) before you assign the local

value, you will get an exception (NameError or UnboundLocalError), depending on which version of Python you are using.

## global Statement

Global variable names can be overridden by local variables if they are declared within the function. Here is another example, similar to the first, but the global and local nature of the variable is not as clear.

```
def foo():
    print "\ncalling foo()…"
    bar = 200
    print "in foo(), bar is", bar
bar = 100
print "in __main__, bar is", bar
foo()
print "\nin __main__, bar is (still)", bar
```

It gave the following output:

```
in __main__, bar is 100
calling foo()…
in foo(), bar is 200
in __main__, bar is (still) 100
```

Our local bar pushed the global bar out of the local scope. To specifically reference a named global variable, one must use the global statement. The syntax for global is:

```
global
                                                    var1[, var2[, …
varN]]]
```

Modifying the example above, we can update our code so that we use the global version of is_this_global rather than create a new local variable.

```
>>> is_this_global = "xyz"
>>> def foo():
…       global is_this_global
…       this_is_local = 'abc'
…       is_this_global = 'def'
…       print this_is_local + is_this_global
…
>>> foo()
```

```
abcdef
>>> print is_this_global
def
```

## Number of Scopes

Python syntactically supports multiple levels of functional nesting; however, a maximum of two scopes is imposed: a function's local scope and the global scope. Even as more levels of functional nesting exist, you are not able to access more than two scopes.

```
def foo():
    m = 3
    def bar():
        n = 4
        print m + n
    print m
    bar()

>>> foo()
Traceback (innermost last):
 File "<interactive input>", line 0, in ?
 File "<interactive input>", line 7, in foo
 File "<interactive input>", line 5, in bar
NameError: m
```

The access to `foo()`'s local variable m within function `bar()` is illegal because m is declared local to `foo()`. The only scopes accessible from `bar()` are `bar()`'s local scope and the global scope. `foo()`'s local scope is *not* included in that short list of two. Note that the output for the "**`print`** `m`" statement succeeded, and it is the function call to `bar()` that fails. (Note: this may change for future versions of Python.)

## Other Scope Characteristics

### Scope and `lambda`

Python's **`lambda`** expressions follow the same scoping rules as standard functions. A **`lambda`** expression defines a new scope, just like a function definition, so the scope is inaccessible to any other part of the program except for that local **`lambda`**/function.

Those **`lambda`** expressions declared local to a function are accessible only within that function; however, the expression in the **`lambda`** statement has the same scope access as the function. In other words, they have access to global variables, but neither has access to each other's local scopes. You can also think of a function and a **`lambda`** expression as siblings.

```
>>> x = 10
>>> def foo():
…       y = 5
…       bar = lambda :x+y
…       print bar()
…       y = 8
…       print bar()
…
>>> foo()
Traceback (innermost last):
 File "<interactive input>", line 0, in ?
 File "<interactive input>", line 4, in foo
 File "<interactive input>", line 3, in <lambda>
NameError: y
```

In the example above, although the **lambda** expression was created in the local scope of `foo()`, it has access to only two scopes: its local scope and the global scope (also see Section 11.8.3). We can correct it by placing a local variable z within the **lambda** expression that references the function local variable y.

```
>>> x = 10
>>> def foo():
…       y = 5
…       bar = lambda z:x+z
…       print bar(y)
…
…       y = 8
…       print bar(y)
…
>>> foo()
15
18
```

### Variable Scope and Namespaces

From our study in this chapter, we can see that at any given time, there are either one or two active scopes—no more, no less. Either we are at the top-level of a module where we have access only to the global scope, or we are executing in a function where we have access to its local scope as well as the global scope. How do namespaces relate to scope?

From the Core Note in Section 11.8.1, we can also see that at any given time, there are either two or three active namespaces. From within a function, the local scope encompasses the local namespace, the first place a name is searched for. If the name exists here, then checking the global scope (global and built-in namespaces) is skipped. From the global scope (outside of any function), a name lookup begins with the global namespace. If no match is found, the search proceeds to the built-in namespace.

We will now present a script with mixed scope everywhere, Example 11.8. We leave it as an exercise to the reader to determine the output of the program.

**Example 11.8. Variable Scope (`scope.py`)**

Local variables hide global variables, as indicated in this variable scope program. What is the output of this program? (And why?)

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2   j, k = 1, 2
003 3
004 4   def proc1():
005 5
006 6       j, k = 3, 4
007 7       print "j == %d and k == %d" % (j, k)
008 8       k = 5
009 9
010 10  def proc2():
011 11
012 12      j = 6
013 13      proc1()
014 14      print "j == %d and k == %d" % (j, k)
015 15
016 16
017 17  k = 7
018 18  proc1()
019 19  print "j == %d and k == %d" % (j, k)
020 20
021 21  j = 8
022 22  proc2()
023 23  print "j == %d and k == %d" % (j, k)
024  <$nopage>
```

## *Recursion

A function is *recursive* if it contains a call to itself. Aho, Sethi, and Ullman define, "[a] procedure is *recursive* if a new activation can begin before an earlier activation of the same procedure has ended." In other words, a new invocation of the same function occurs within that function before it finished.

Recursion is used extensively in language recognition as well as in mathematical applications that use recursive functions. Earlier in this text, we took a first look at the factorial function where we defined:

```
N! = factorial(N) = 1 * 2 * 3 … * N
```

We can also look at factorial this way:

```
factorial(N)  = N!
              = N * (N-1)!
              = N * (N-1) * (N-2)!

              = N * (N-1) * (N-2) … * 3 * 2 * 1
```

We can now see that factorial is recursive because `factorial(N) = N * factorial(N-1)`. In other words, to get the value of `factorial(N)`, one needs to calculate `factorial(N-1)`. Furthermore, to find `factorial(N-1)`, one needs to computer `factorial(N-2)`, and so on.

We now present the recursive version of the factorial function:

```
def factorial(n):
    if n == 0 or n == 1:# 0! = 1! = 1
        return 1
    else:
        return (n * factorial(n-1))
```

## Exercises

**1:**

*Arguments*. Compare the following three functions:

```
def countToFour1():
    for eachNum in range(5):
        print eachNum,

def countToFour2(n):
    for eachNum in range(n, 5):
        print eachNum,

def countToFour3(n=1):
    for eachNum in range(n, 5):
        print eachNum,
```

What do you think will happen as far as output from the program, given the following input values? Enter the output into Table 11.1 below. Write in "ERROR" if you think one will occur with the given input or " NONE" if there is no output.

**Table 11.2. Output chart for Problem 11-1**

| *Input* | countToFour1 | countToFour2 | countToFour3 |
|---|---|---|---|
| 2 | | | |

| | | | |
|---|---|---|---|
| 4 | | | |
| 5 | | | |
| ***(nothing)*** | | | |

**2:**

*Functions.* Combine your solutions for Exercise 5-2. such that you create a combination function which takes the same pair of numbers and returns both their sum and product at the same time.

**3:**

*Functions.* In this exercise, we will be implementing the `max()` and `min()` built-in functions.

(a) Write simple functions `max2()` and `min2()` that take two items and return the larger and smaller item, respectively. They should work on arbitrary Python objects. For example, `max2(4, 8)` and `min2(4, 8)` would each return 8 and 4, respectively.

(b) Create new functions `my_max()` and `my_min()` that use your solutions in part (a) to recreate `max()` and `min()`. These functions return the largest and smallest item of non-empty sequences, respectively. Test your solutions for numbers and strings.

**4:**

*Return values.* Create a complementary function to your solution for Exercise 5-13. Create a function that takes a total time in minutes and returns the equivalent in hours and minutes.

**5:**

*Default arguments.* Update the sales tax script you created in Exercise 5-7 such that a sales tax rate is no longer required as input to the function. Create a default argument using your local tax rate if one is not passed in on invocation.

**6:**

*Variable-length arguments.* Write a function called `printf()`. There is one positional argument, a format string. The rest are variable arguments that need to be displayed to standard output based on the values in the format string, which allows the special string format operator directives such as `%d`, `%f`, etc. HINT: the solution is trivial—there is no need to implement the string operator functionality, but you do need to use the string format operator ( % ) explicitly.

**7:**

*Functional programming with* `map()`. Given a pair of identically-sized lists, say `[1, 2, 3, ...]`, and `['abc', 'def', 'ghi', ...]`, merge both lists into a single list consisting of tuples of elements of each list so that our result looks like: `{[(1, 'abc'), (2, 'def'), (3, 'ghi'), ...}`. (Although this problem is similar in nature to a problem in Chapter 6, there is no direct correlation between their solutions.)

**8:**

*Functional programming with* `filter()`. Use the code you created for Exercise 5-4 to determine leap years. Update your code so that it is a function if you have not done so already. Then write some code to take a list of years and return a list of only leap years.

**9:**

*Functional programming with* `reduce()`. Review the code in Section 11.7.2 that illustrated how to sum up a set of numbers using `reduce()`. Modify it to create a new function called `average()` that calculates the simple average of a set of numbers.

**10:**

*Functional programming with* `filter()`. In the Unix file system, there are always two special files in each folder/directory: '.' indicates the current directory and '..' represents the parent directory. Given this knowledge, take a look at the documentation for the `os.listdir()` function and describe what this code snippet does:

```
files = filter(lambda x: x and x[0] != '.', os.listdir(folder))
```

**11:**

*Functional programming with* `map()`. Write a program that takes a file name and "cleans" the file by removing all leading and trailing whitespace from each line. Read in the original file and write out a new one, either creating a new file or overwriting the existing one. Give your user the option to pick which of the two to perform.

**12:**

*Passing functions.* Write a sister function to the `testit()` function described in this chapter. Rather than testing execution for errors, `timeit()` will take a function object (along with any arguments), and time how long it takes to execute the function. Return the following values: function return value, time elapsed. You can

use `time.clock()` or `time.time()`, whichever provides you with greater accuracy.

**13:**

*Functional programming with `reduce()` and Recursion.* In Chapter 8, we looked at N factorial or N! as the product of all numbers from 1 to N.

(a)Take a minute to write a small, simple function called `mult(x, y)` that takes `x` and `y` and returns their product.

(b) Use the `mult()` function you created in part (a)along with `reduce()` to calculate factorials.

(c) Discard the use of `mult()` completely and use a **`lambda`** expression instead.

(d) In this chapter, we presented a recursive solution to finding N! Use the `timeit()` function you completed in the problem above and time all three versions of your factorial function (iterative, `reduce()`, and recursive). Explain any differences in performance, anticipated and actual.

**14:**

**Recursion.* We also looked at Fibonacci numbers in Chapter 8. Rewrite your previous solution for calculating Fibonacci numbers (Exercise 8-9) so that it now uses recursion.

**15:**

**Recursion.* Rewrite your solution to Exercise 6-5 which prints a string backwards to use recursion. Use recursion to print a string forward *and* backward.

# Chapter 12. Modules

This chapter focus on Python modules and how data is imported from modules into your programming environment. We will also take a look at packages. Modules are a way to organize Python code, and packages help you organize modules. We will conclude this chapter with a look at other related aspects of modules.

## What are Modules?

A *module* allows you to logically organize your Python code. When code gets to be large enough, the tendency is to break it up into organized pieces which can still interact with each other at a functioning level. These pieces generally have attributes which have some relation to each other, perhaps a single class with its member data variables and methods, or maybe a group of related, yet independently operating functions. These pieces should be shared, so Python allows a module the ability to "bring in" and use attributes from other modules to take advantage of work that has been done, maximizing code reusability. This process of associating attributes from other modules with your module is called *importing*. Self-contained and organized pieces of Python code that can be shared—in a nutshell, that describes a module.

## Modules and Files

If modules represent a logical way to organize your Python code, then files are a way to physically organize modules. To that end, each file is considered an individual module, and vice versa. The file name of a module is the module name appended with the `.py` file extension. There are several aspects we need to discuss with regards to what the file structure means to modules. Unlike other languages in which you import classes, in Python you import modules or module attributes.

### Namespaces

We will discuss namespaces in detail later in this chapter, but the basic concept of a namespace is an individual set of mappings from names to objects. As you are no doubt aware, module names play an important part in the naming of their attributes. The name of the attribute is always prepended with the module name. For example, the `atoi()` function in the `string` module is called `string.atoi()`. Because only one module with a given name can be loaded into the Python interpreter, there is no intersection of names from different modules; hence, each module defines its own unique namespace. If I created a function called `atoi()` in my own module, perhaps `mymodule`, its name would be `mymodule.atoi()`. So even if there is a name conflict for an attribute, the *fully-*

*qualified name*—referring to an object via dotted attribute notation—prevents an exact and conflicting match.

## Search Path and Path Search

The process of importing a module requires a process called a *path search*. This is the procedure of checking "predefined areas" of the file system to look for your `mymodule.py` file in order to load the `mymodule` module. These predefined areas are no more than a set of directories that are part of your Python *search path*. To avoid the confusion between the two, think of a path search as the pursuit of a file through a set of directories, the search path.

There may be times where importing a module fails:

```
>>> import xxx
Traceback (innermost last):
 File "<interactive input>", line 1, in ?
ImportError: No module named xxx
```

When this error occurs, the interpreter is telling you it cannot access the requested module, and the likely reason is that the module you desire is not in the search path, leading to a path search failure.

A default search path is automatically defined either in the compilation or installation process. This search path may be modified in one of two places.

One is the `PYTHONPATH` environment variable set in the *shell* or command-line interpreter that invokes Python. The contents of this variable consist of a colon-delimited set of directory paths. If you want the interpreter to use the contents of this variable, make sure you set or update it before you start the interpreter or run a Python script.

Once the interpreter has started, you can access the path itself which is stored in the `sys` module as the `sys.path` variable. Rather than a single string that is colon-delimited, the path has been "split" into a list of individual directory strings. Below is an example search path for a Unix machine. Your mileage will definitely vary as you go from system to system.

```
>>> sys.path
['', '/usr/local/lib/python1.5/', '/usr/local/lib/
python1.5/plat-sunos5', '/usr/local/lib/python1.5/
lib-tk', '/usr/local/lib/python1.5/lib-dynload']
```

Bearing in mind that this is just a list, we can definitely take our liberty with it and modify it at our leisure. If you know of a module you want to import, yet its directory is

not in the search path, by all means use the list's `append()` method to add it to the path, like so:

```
sys.path.append('/home/wesc/py/lib')
```

Once this is accomplished, you can then load your module. As long as one of the directories in the search path contains the file, then it will be imported. Of course, this adds the directory only to the end of your search path. If you want to add it elsewhere, such as in the beginning or middle, then you have to use the `insert()` list method for those. In our examples above, we are updating the `sys.path` attribute interactively, but it will work the same way if run as a script.

Here is what it would look like if we ran into this problem interactively:

```
>>> import sys
>>> import mymodule
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named mymodule
>>>
>>> sys.path.append('/home/wesc/py/lib')
>>> sys.path
['', '/usr/local/lib/python1.5/', '/usr/local/lib/
python1.5/plat-sunos5', '/usr/local/lib/python1.5/
lib-tk', '/usr/local/lib/python1.5/lib-dynload', '/home/
wesc/py/lib']
>>>
>>> import mymodule
>>>
```

On the flip side, you may have too many copies of a module. In the case of duplicates, the interpreter will load the first module it finds with the given name while rummaging through the search path in sequential order.

## Namespaces

A *namespace* is a mapping of names (identifiers) to objects. The process of adding a name to a namespace consists of *binding* the identifier to the object (and increasing the reference count to the object by one). The Python Language Reference also includes the following definitions: "changing the mapping of a name is called *rebinding*[, and] removing a name is *unbinding*."

As briefly introduced in the last chapter, there are either two or three active namespaces at any given time during execution. These three namespaces are the local, global, and built-ins namespaces, but local namespaces come and go during execution, hence the

"two or three" we just alluded to. The names accessible from these namespaces as dependent on their *loading order,* or the order in which the namespaces are brought into the system.

The Python interpreter loads the built-ins namespace first. This consists the names in the `__builtins__` module. Then the global namespace for the executing module is loaded, which then becomes the active namespace when the module begins execution. Thus we have our two active namespaces.

**NOTE**

The `__builtins__` module should not be confused with the `__builtin__` module. The names, of course, are so similar that it tends to lead to some confusion among new Python programmers who have gotten this far. The `__builtins__` module consists of a set of built-in names for the built-ins namespace. Most, if not all, of these names come from the `__builtin__` module, which is a module of the built-in functions, exceptions, and other attributes. In standard Python execution, `__builtins__` contains all the names from `__builtin__`. The only time there is a difference is when executing in restricted mode. (Restricted execution is covered formally in *Chapter 14*.) In restricted mode, `__builtins__` only consists of a subset of the attributes from `__builtin__` which can be accessed from within a restricted environment.

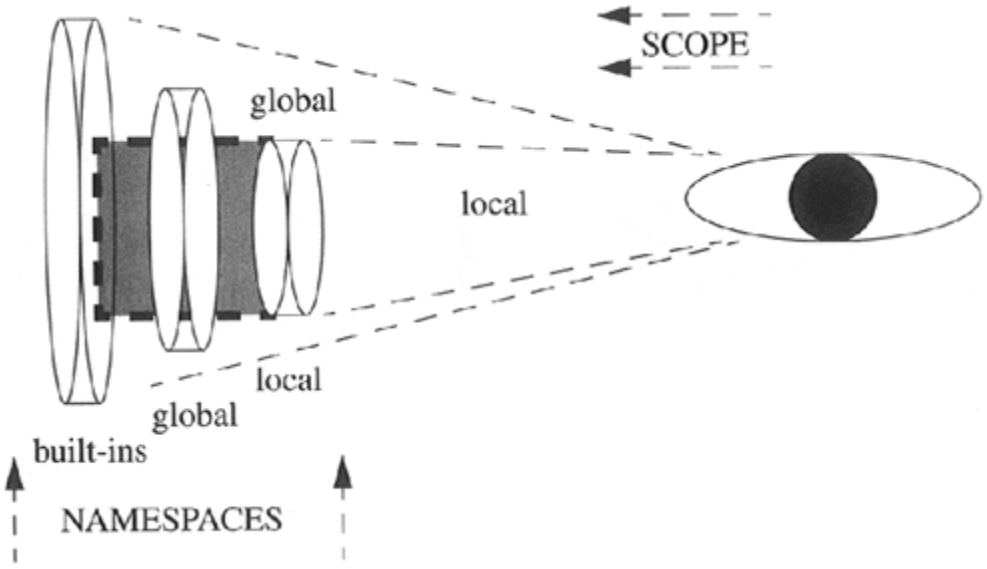
When a function call is made during execution, the third, a local, namespace is created. We can use the `globals()` and `locals()` built-in functions to tell us which names are in which namespaces. We will discuss both functions in more detail later on in this chapter.

## Namespaces vs. Variable Scope

Okay, now that we know what namespaces are, how do they relate to variable scope again? They seem extremely similar. The truth is, you are quite correct.

Namespaces are purely mappings between names and objects, but scope dictates how or rather, where, one can access these names based on the physical location from within your code. We illustrate the relationship between namespaces and variable scope in Figure 12-1.

**Figure 12.1. Namespaces vs. Variable Scope**




Notice that each of the namespaces is a self-contained unit. But looking at the namespaces from the scoping point of view, things appear different. All names within the local namespace are within my local scope. Any name outside my local scope is in my global scope.

Also keep in mind that during the execution of the program, the local namespaces and scope are transient because function calls come and go, but the global and built-ins namespaces remain.

Our final thought to you in this section is, when it comes to namespaces, ask yourself the question, "Does it have it?" And for variable scope, ask, "Can I see it?"

## Name Lookup, Scoping, and Overriding

So how do scoping rules work in relationship to namespaces? It all has to do with name lookup. When accessing an attribute, the interpreter must find it in one of the three namespaces. The search begins with the local namespace. If the attribute is not found there, then the global namespace is searched. If that is also unsuccessful, the final frontier is the built-ins namespace. If the exhaustive search fails, you get the familiar:

```
>>> foo
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: foo
```

Notice how the figure features the foremost-searched namespaces "shadowing" namespaces which are searched afterwards. This is to try to convey the effect of *overriding*. This is the process whereby names may be taken out-of-scope because a more

local namespace contains a name. Take a look at the following piece of code that was introduced in the previous chapter:

```
def foo():
    print "\ncalling foo()…"
    bar = 200
    print "in foo(), bar is", bar

bar = 100
print "in __main__, bar is", bar
foo()
```

When we execute this code, we get the following output:

```
in __main__, bar is 100

calling foo()…
in foo(), bar is 200
```

The `bar` variable in the local namespace of `foo()` overrode the global `bar` variable. Although `bar` exists in the global namespace, the lookup found the one in the local namespace first, hence "overriding" the global one. For more information regarding scope, please go back to Section 11.8 in the last chapter.

## Importing Modules

Importing a module requires the use of the **`import`** statement, whose syntax is:

```
import module1[, module2[, … moduleN]]
```

When this statement is encountered by the interpreter, the module is imported if found in the search path. Scoping rules apply, so if imported from the top-level of a module, it has global scope; if imported from a function, it has local scope.

When a module is imported the first time, it is loaded and executed.

### Module "Executed" When Loaded

One effect of loading a module is that the imported module is "executed," that is, the top-level portion of the imported module is directly executed. This usually includes setting up

of global variables as well as performing the class and function declarations, and if there is a check for `__name__` to do more on direct script invocation, that is executed too.

Of course, this type of execution may or may not be the desired effect. If not, you will have to put as much code as possible into functions. Suffice it to say that good module programming style dictates that only function and/or class definitions should be at the top-level of a module.

### Importing vs. Loading

A module is loaded only once, regardless of the number of times it is imported. This prevents the module "execution" from happening over and over again if multiple imports occur. If your module imports the `sys` module, and so do five of the other modules you import, it would not be wise to load `sys` (or any other module) each time! So rest assured, loading happens only once, on first import.

## Importing Module Attributes

It is possible to import specific module elements into your own module. By this, we really mean importing specific names from the module into the current namespace. For this purpose, we can use the **`from-import`** statement, whose syntax is:

```
from module import name1[, name2[, … nameN]]
```

### Names Imported into Current Namespace

Calling **`from-import`** brings the name into the current namespace, meaning that you do not use the attribute/dotted notation to access the module identifier. For example, to access a variable named *`var`* in module *`module`* that was imported with:

```
from module import var
```

we would use *`var`* by itself. There is no need to reference the module since you just imported.

It is also possible to import all the names from the module into the current namespace using the following **`from-import`** statement:

**from** *module* **import ***

**NOTE**

In practice, using **from** *module* **import *** is considered poor style because it "pollutes" the current namespace and has the potential of overriding names in the current namespace; however, it is extremely convenient if a module has many variables which are often accessed, or if the module has a very long name.

We recommend using this form in only two situations. The first is where the target module has many attributes that would make it inconvenient to type in the module name over and over again. Two prime examples of this are the `Tkinter` (Python/Tk) and `NumPy` (Numeric Python) modules, and perhaps the `socket` module. The other place where it is acceptable to use **from** *module* **import *** is within the interactive interpreter, to save on the amount of typing.

## Names Imported into Importer's Scope

Another side effect of importing only names from other modules is that the names are now part of the scope of the importing module.This means that changes to the variable affect only the local copy and not the original in the imported module's namespace. In other words, the binding is now local rather than across namespaces.

Below, we present the code to two modules: an importer, `impter.py`, and an importee, `imptee.py`. Currently, `imptr.py` uses the **from-import** statement which creates only local bindings.

```
#############
# imptee.py #
#############
foo = 'abc'
def show():
    print 'foo from imptee:', foo
```

```
#############
# impter.py #
#############
from imptee import foo, show
show()
foo = 123
print 'foo from impter:', foo
```

```
show()
```

Upon running the importer, we discover that the importee's view of its `foo` variable has not changed even though we modified it in the importer.

```
foo from imptee: abc
foo from impter: 123
foo from imptee: abc
```

The only solution is to use import and *fully-qualified* identifier names using the attribute/dotted notation.

```
#############
# impter.py #
#############
import imptee
imptee.show()
imptee.foo = 123
print 'foo from impter:', imptee.foo
imptee.show()
```

Once we make the update and change our references accordingly, we now have achieved the desired effect.

```
foo from imptee: abc
foo from impter: 123
foo from imptee: 123
```

## Module Built-in Functions

The importation of modules has some functional support from the system. We will look at those now.

### `__import__()`

The `__import__()` function is new as of Python 1.5, and it is the function that actually does the importing, meaning that the import statement invokes the `__import__()` function to do its work. The purpose of making this a function is to allow for overriding it if one is inclined to develop his or her own importation algorithm.

The syntax of `__import__()` is:

```
__import__(module_name[, globals[, locals[, fromlist]]]
```

The *module_name* variable is the name of the module to import, *globals* is the dictionary of current names in the global symbol table, *locals* is the dictionary of current names in the local symbol table, and *fromlist* is a list of symbols to import the way they would be imported using the **from-import** statement.

The globals, locals, and fromlist arguments are optional, and if not provided, default to globals(), locals(), and [], respectively.

Calling '**import** sys' can be accomplished with

```
sys = __import__('sys')
```

## globals() and locals()

The globals() and locals() built-in functions return dictionaries of the global and local namespaces, respectively, of the caller. From within a function, the local namespace represents all names defined for execution of that function, which is what locals() will return. globals(), of course, will return those names globally accessible to that function.

From the global namespace, however, globals() and locals() return the same dictionary because the global namespace is as local as you can get while executing there. Here is a little snippet of code that calls both functions from both namespaces:

```
def foo():
    print '\ncalling foo()…'
    aString = 'bar'
    anInt = 42
    print "foo()'s globals:", globals().keys()
    print "foo()'s locals:", locals().keys()

print "__main__'s globals:", globals().keys()
print "__main__'s locals:", locals().keys()
foo()
```

We are going to ask for the dictionary keys only because the values are of no consequence here (plus they make the lines wrap even more in this text). Executing this script, we get the following output:

```
% namespaces.py
```

```
__main__'s globals: ['__doc__', 'foo', '__name__',
'__builtins__']
__main__'s locals: ['__doc__', 'foo', '__name__',
'__builtins__']

calling foo()…
foo()'s globals: ['__doc__', 'foo', '__name__',
'__builtins__']
foo()'s locals: ['anInt', 'aString']
```

### reload()

The `reload()` built-in function performs another import on a previously imported module. The syntax of `reload()` is:

```
reload(module)
```

*`module`* is the actual module you want to reload. There are some criteria to using the `reload()` module. The first is that the module must have been imported in full (not by using **`from-import`**), and it must have loaded successfully. The second rule follows from the first, and that is the argument to `reload()` is the module itself and not a string containing the module name, i.e., it must be something like `reload(sys)` instead of `reload('sys')`.

Also, code in a module is executed when it is imported, but only once. A second import does not re-execute the code, it just binds the module name. Thus `reload()` makes sense, as it overrides this default behavior.

## Packages

A *package* is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages. Packages were added to Python 1.5 to aid with a variety of problems including:

Adding hierarchical organization to flat namespace

Allowing developers to group-related modules

Allowing distributors to ship directories vs. bunch of files

Helping resolve conflicting module names

Along with classes and modules, packages use the familiar attribute/dotted attribute notation to access their elements. Importing modules within packages use the standard **`import`** and **`from-import`** statements.

## Directory Structure

The example package directory structure below is available for you to experiment with on the CD-ROM accompanying this book. Just browse the code samples and navigate to Chapter 12.

```
Phone/
    __init__.py
    Voicedta/
        __init__.py
        Pots.py
        Isdn.py
    Fax/
        __init__.py
        G3.py
    Mobile/
        __init__.py
        Analog.py
        Digital.py
    Pager/
        __init__.py
        Numeric.py
```

`Phone` is top-level package and `Voicedta,` etc., are subpackages. Import subpackages by using **`import`** like this:

```
import Phone.Mobile.Analog

Phone.Mobile.Analog.dial()
```

Alternatively, you can use **`from-import`** in a variety of ways:

The first way is importing just the top-level subpackage and referencing down the subpackage tree using the attribute/dotted notation:

```
from Phone import Mobile

Mobile.Analog.dial('4 555-1212')
```

Further more, we can do down one more subpackage for referencing:

```
from Phone.Mobile import Analog

Analog.dial('555-1212')
```

In fact, you can go all the way down in the subpackage tree structure:

```
from Phone.Mobile.Analog import dial

dial('555-1212')
```

In our above directory structure hierarchy, we observe a number of `__init__.py` files. These are initializer modules that are required when using **`from-import`** to import subpackages, but should otherwise exist though they can remain empty.

### Using `from-import` with Packages

Packages also support the **`from-import`** all statement:

```
from
                                        package.module
                                        import *
```

However, such a statement is too operating system filesystem-dependent for Python to make the determination which files to import. Thus the `__all__` variable in `__init__.py` is required. This variable contains all the module names that should be imported when the above statement is invoked if there is such a thing. It consists of a list of module names as strings.

## Other Features of Modules

### Auto-loaded Modules

When the Python interpreter starts up in standard mode, some modules are loaded by the interpreter for system use. The only one that affects you is the `__builtin__` module, which normally gets loaded in as the `__builtins__` module.

The `sys.modules` variable consists of a dictionary of modules that the interpreter has currently loaded (in full and successfully) into the interpreter. The module names are the keys, and the location from which they were imported are the values.

For example, in Windows, the `sys.modules` variable contains a large number of loaded modules, so we will shorten the list by requesting only the module names. This is accomplished by using the dictionary's `keys()` method:

```
>>> import sys
>>> sys.modules.keys()
['os.path', 'os', 'exceptions', '__main__', 'ntpath',
'strop', 'nt', 'sys', '__builtin__', 'site',
'signal', 'UserDict', 'string', 'stat']
```

The loaded modules for Unix are quite similar:

```
>>> import sys
>>> sys.modules.keys()
['os.path', 'os', 'readline', 'exceptions',
'__main__', 'posix', 'sys', '__builtin__', 'site',
'signal', 'UserDict', 'posixpath', 'stat']
```

### Preventing Attribute Import

If you do not want module attributes imported when a module is imported with "**from** *module* **import ***", begin the name, and prepend the underscore ( _ ) to their names. Names in the imported module that begin with an underscore ( _ ) are not imported. This minimal level of data hiding does not apply if the entire module is imported.

## Exercises

**1:**

*PathSearch vs. SearchPath.* What is the difference between a path search and a search path?

**2:**

*Importing Attributes.* Assume you have a function called `foo()` in your module `mymodule`. What are the two ways of import this function into your namespace for invocation?

**3:**

*Importing.* What are the differences between using "**import** *module*" and "**from** *module* **import ***"?

**4:**

*Namespaces vs. Variable Scope.* How are namespaces and variable scopes different from each other?

**5:**

Using *__import__()*.

(a) Use `__import__()` to import a module into your namespace. What is the correct syntax you finally used to get it working?

(b) Same as above, but use `__import__()` to import only specific names from modules.

**6:**

*Extended Import.* Create a new function called `importAs()`. This function will import a module or module into your namespace, but with a name you specify, not its original name. For example, calling `newname=importAs('mymodule')`, will import the module `mymodule`, but the module and all its elements are accessible only as `newname` or `newname.attr`. You will discover that this is the exact functionality provided by the new extended import syntax introduced in Python 2.0.

# Chapter 13. Classes and OOP

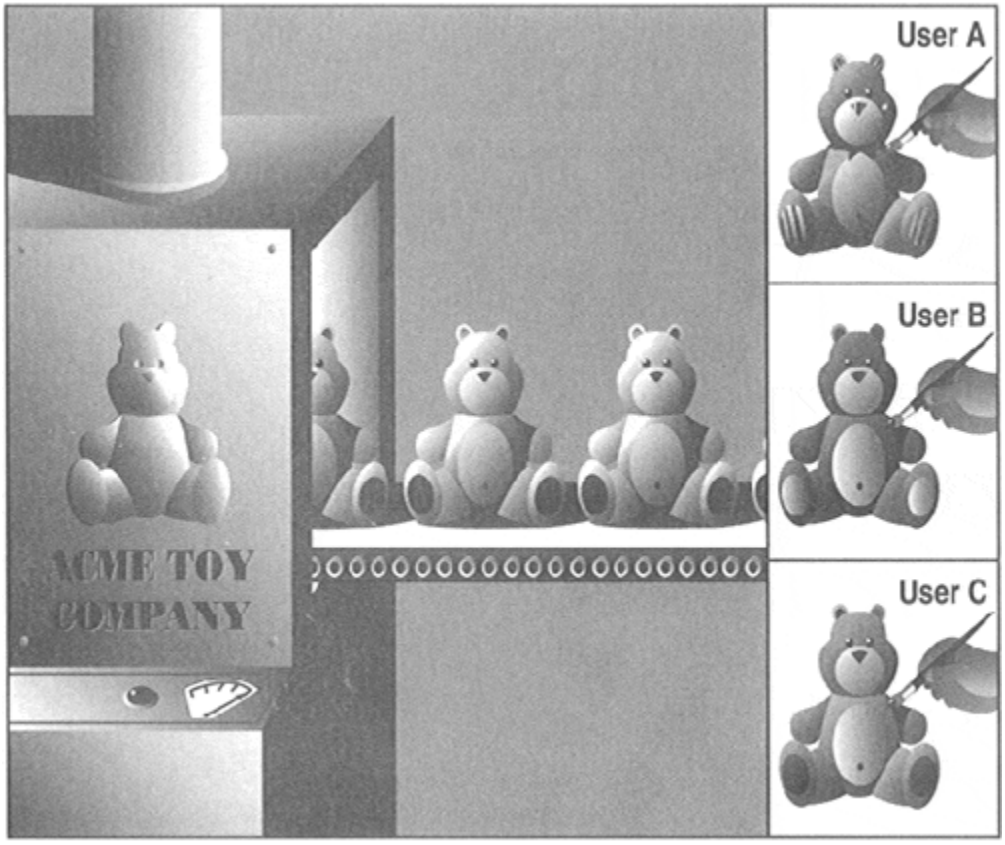
Classes finally introduce the notion of object-oriented programming (OOP) to our picture. We will first present a high-level overview, covering all the main aspects of using classes and OOP in Python. The remainder of the chapter covers all the details on classes, class instances, and methods. We will also describe derivation or subclassing in Python and what its inheritance model is. Finally, Python provides special attributes which allow the programmer to customize classes with special functionality, including those which overload operators and emulate Python types. We will show you how to implement some of these special methods to customize your class to attain type-like behavior.

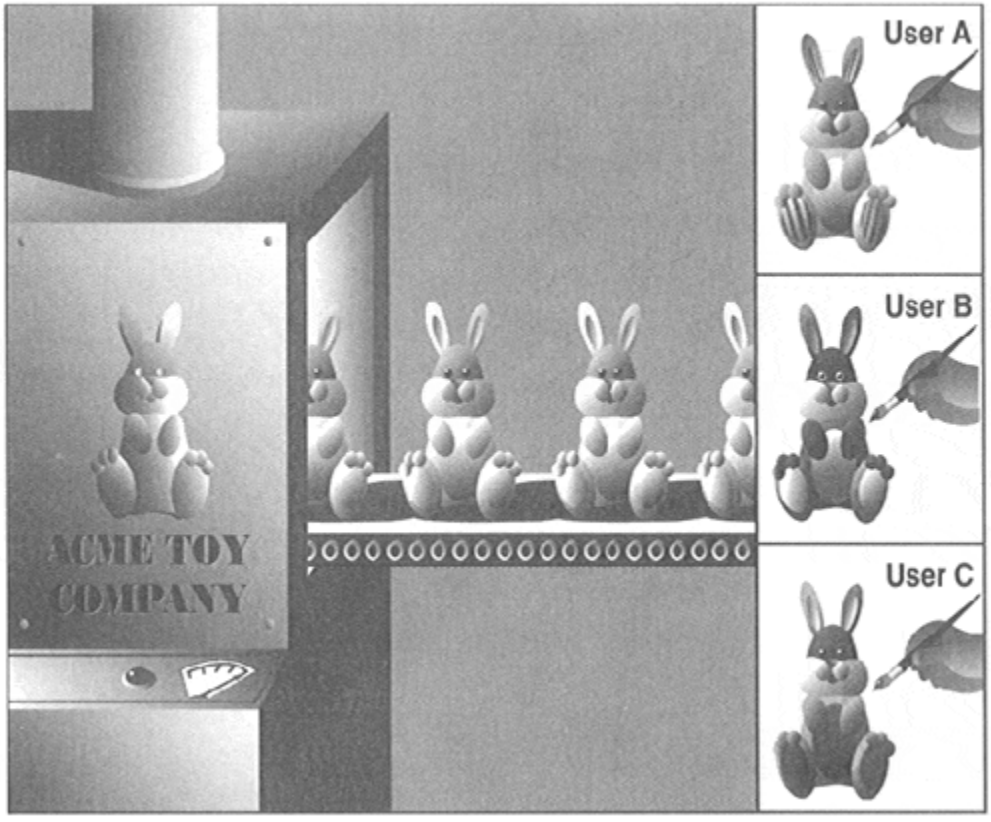
## Introduction

Before we get into the nitty-gritty of OOP and classes, we begin with a high-level overview, then present some simple examples to get you "warmed up." If you are new to object-oriented programming, you may wish to merely skim this section first, then begin the formal reading in Section 13.2. This section is targeted more to those already familiar with the concepts, who simply want to see "how it's done" in Python.

The main two entities in Python object-oriented programming are classes and class instances (see Figure13-1).

**Figure 13.1. The factory manufacturing machines on the left are analogous to classes, while each toy produced are instances of their respective classes. Although each instance has the basic underlying structure, individual attributes like color or feet can be changed—these are similar to instance attributes.**


User A
User B
User C
ACME TOY
COMPANY


User A
User B
User C
ACME TOY
COMPANY

## Classes and Instances

Classes and instances are related to each other: classes provide the definition of an object, and instances are "the real McCoy," the objects specified in the class definition brought to life.

Here is an example of how to create a class:

```
class MyNewObjectType:
    'define MyNewObjectType class'
   class_suite
```

The keyword is **class,** followed by the class name. What follows is the suite of code that defines the class. This usually consists of various definitions and declarations. The process of creating an instance is called *instantiation,* and it is carried out like this (note the conspicuous absence of a `new` keyword):

```
myFirstObject = MyNewObjectType()
```

The class name is given as an "invocation," using the familiar function operators ( `( )` ). You then typically assign that newly-created instance to a variable. The assignment is not required syntactically, but if you do not save your instance to a variable, it will be of no use and will be automatically garbage-collected because there would no references to that instance. What you would be doing is allocating memory, then immediately deallocating it.

Classes can be as simple or as complicated as you wish them to be. At a very minimum, classes can be used as "namespace containers." By this, we mean that you can store data into variables and group them in such a way that they have all share the same relationship—a named relationship using the standard Python dotted-attribute notation. For example, you may have a class without any inherent attributes and merely use such a class to provide a namespace for data, giving your class characteristics similar to records in Pascal or structures in C, or, in other words, use the class as simply a container object with shared naming.

Here is one example of such a class:

```
class MyData:
    pass
```

Recall that the **pass** statement is used where code is required syntactically, but no operation is desired. In this case, the required code is the class suite, but we do not wish to provide one. The class we just defined has no methods or any other attributes. We will now create an instance to use the class simply as a namespace container.

```
>>> mathObj = MyData()
>>> mathObj.x = 4
>>> mathObj.y = 5
>>> mathObj.x + mathObj.y
9
>>> mathObj.x * mathObj.y
20
```

We could have used variables "x" and "y" to accomplish the same thing, but in our case, `mathObj.x` and `mathObj.y` are related by the instance name, `mathObj`. This is what we mean by using classes as namespace containers. `mathObj.x` and `mathObj.y` are known as *instance attributes* because they are only attributes of their instance object (`mathObj`), not of the class (`MyData`).

## Methods

One way we can improve our use of classes is to add functions to them. These class functions are known by their more common name, *methods*. In Python, methods are defined as part of the class definition, but can be invoked only by an instance. In other words, the path one must take to finally be able to call a method goes like this: (1) define the class (and the methods), (2) create an instance, and finally, (3) invoke the method from that instance. Here is an example class with a method:

```
class MyDataWithMethod:         # define the class
    def printFoo(self):         # define the method
        print 'You invoked printFoo()!'
```

You will notice the `self` argument, which must be present in all method invocations. That argument, representing the instance object, is passed to the method implicitly by the interpreter when you invoke a method via an instance, so you, yourself, do not have to worry about passing anything in. Now we will instantiate the class and invoke the method once we have an instance:

```
>>> myObj = MyDataWithMethod()  # create the instance
>>> myObj.printFoo()            # now invoke the method
You invoked printFoo()!
```

We conclude this introductory section by giving you a slightly more complex example of what you can do with classes (and instances) and also introducing you to the special method `__init__()` as well as subclassing and inheritance.

For those of you who are already familiar with object-oriented programming, `__init__()` is the *class constructor*. If you are new to the world of OOP, a constructor is simply a special method which is called during instantiation that defines additional behavior that should occur when a class is instantiated, i.e., setting up initial values or running some preliminary diagnostic code—basically performing any special tasks or setup after the instance is created but before it is returned from the instantiation call.

(We will add **`print`** statements to our methods to better illustrate when certain methods are called. It is generally not typical to have input or output statements in functions unless output is a predetermined characteristic of the body of code.)

## Creating a Class (Class Definition)

```
class AddrBookEntry:                    # class definition
  'address book entry class'
  def __init__(self, nm, ph):           # define constructor
    self.name = nm                      # set inst .attr .1
    self.phone = ph                     # set inst .attr .2
    print 'Created instance for:', self.name

  def updatePhone(self, newph):         # define method
    self.phone = newph
    print 'Updated phone# for:', self.name
```

In the definition for the `AddrBookEntry` class, we define two methods: `__init__()` and `updatePhone()`. `__init__()` is called when instantiation occurs, that is, when `AddrBookEntry()` is invoked. You can think of such an instantiation call to be an implicit call to `__init__()` because the arguments given in the call to AddrBookEntry() are exactly the same as those that are received by `__init__()`.

Recall that the `self` (instance object) argument is passed in automatically by the interpreter when the method is invoked from an instance, so in our `__init__()` above, the only required arguments are `nm` and `ph`, representing the name and telephone number, respectively. `__init__()` sets these two instance attributes on instantiation so that they are available to the programmer by the time the instance is returned from the instantiation call.

As you may have surmised, the purpose of the `updatePhone()` method is to replace an address book entry's telephone number attribute.

## Creating Instances (Instantiation)

```
>>> john = AddrBookEntry('John Doe', '408-555-1212')
Created instance for: John Doe
>>> jane = AddrBookEntry('Jane Doe', '650-555-1212')
Created instance for: Jane Doe
```

These are our instantiation calls which, in turn, invoke `__init__()`. Recall that an instance object is passed in automatically as `self`. So in your head, you can replace `self` in methods with the name of the instance. In the first case, when object `john` is instantiated, it is `john.name` that is set, as you can confirm below.

Also, without the presence of default arguments, both parameters to `__init__()` are required as part of the instantiation invocation.

## Accessing Instance Attributes

```
>>> john
<__main__.AddrBookEntry instance at 80ee610>
>>> john.name
'John Doe'
>>> john.phone
'408-555-1212'
>>> jane.name
'Jane Doe'
>>> jane.phone
'650-555-1212'
```

Once our instance was created, we can confirm that our instance attributes were indeed set by `__init__()` during instantiation. Calling the instance within the interpreter tells us what kind of object it is. (We will discover later how we can customize our class so that rather than seeing the default `<...>` Python object string, a more desired output can be customized.)

## Method Invocation (via Instance)

```
>>> john.updatePhone('415-555-1212')
Updated phone# for: John Doe
>>> john.phone
'415-555-1212'
```

The `updatePhone()` method requires one explicit argument: the new phone number. We check our instance attribute right after the call to `updatePhone()`, making sure that it did what was advertised.

So far, we have invoked only a method via an instance, as in the above example. These are known as *bound methods* in Python. Binding is just a Python term to indicate whether we have an instance to invoke a method.

## Creating a Subclass

Subclassing with inheritance is a way to create and customize a new class type with all the features of an existing class but without modifying the original class definition. The new *subclass* can be customized with special functionality unique only to that new class type. Aside from its relationship to its *parent* or *base class,* a subclass has all the same features as any regular class and is instantiated in the same way as all other classes. Note below that a parent class is part of the subclass declaration:

```
class AddrBookEntryWithEmail(AddrBookEntry):     # define subclass
    'update address book entry class'
    def __init__(self, nm, ph, em):              # new __init__
      AddrBookEntry.__init__(self, nm, ph)       # base class cons.
      self.email = em
    def updateEmail(self, newem):                # define method
        self.email = newem
        print 'Updated e-mail address for:', self.name
```

We will now create our first subclass, `AddrBookEntryWithEmail`. In Python, when classes are derived, subclasses inherit the base class attributes, so in our case, we will not only define the methods `__init__()` and `updateEmail()`, but `AddrBookEntryWithEmail` will also inherit the `updatePhone()` method from `AddrBookEntry`.

Each subclass must define its own constructor if desired, otherwise, the base class constructor will be called. However, if a subclass overrides a base class constructor, the base class constructor will *not* be called automatically—such a request must be made explicitly as we have above. For our subclass, we make an initial call to the base class constructor before performing any "local" tasks, hence the call to `AddrBookEntry.__init__()` to set the name and phone number. Our subclass sets one additional instance attribute, the e-mail address, which is set by the remaining line of our constructor.

Note how we have to explicitly pass the `self` instance object to the base class constructor because we are not invoking that method from an instance. We are invoking that method from an instance of a subclass. Because we are not invoking it via an instance, this *unbound method* call requires us to pass an acceptable instance (`self`) to the method.

We close out this section with examples of how to create an instance of the subclass, accessing its attributes, and invoking its methods, including those inherited from the parent class.

## Using a Subclass

```
>>> john = AddrBookEntryWithEmail('John Doe, '408-555-
1212', 'john@spam.doe')
Created instance for: John Doe
>>> john
<__main__.AddrBookEntryWithEmail instance at 80ef6f0>
>>> john.name
'John Doe'
>>> john.phone
'408-555-1212'
>>> john.email
'john@spam.doe'
>>> john.updatePhone('415-555-1212')
Updated phone# for: John Doe
>>> john.phone
'415-555-1212'
>>> john.updateEmail('john@doe.spam')
Updated e-mail address for: John Doe
>>> john.email
'john@doe.spam'
```

**NOTE**

Class names traditionally being with a capital letter. This is the standard convention that will help you identify classes, especially during instantiation (which would look like a function call otherwise). In particular, data attributes should sound like data value names, and methods names should indicate action towards a specific object or value. Another way to phrase this is: Use nouns for data value names and predicates (verbs plus direct objects) for methods. The data items are the objects you, the programmer, are acting on, and the methods should indicate what action the programmer wants to perform on the object.

In the classes we defined above, we attempted to follow this guideline, with data values such as "name," "phone," and "email," and actions such as "updatePhone" and "updateEmail." Other good examples for values include "data," "amount," or "balance;" some recommended method names include "getValue," "setValue," and "clearDataset." Classes should also be well named; some of those good names include "AddrBookEntry," "RepairShop," etc.

We hope that you now have some understanding of how object-oriented programming is accomplished using Python. The remaining sections of this chapter will take you deeper into all the facets of object-oriented programming and Python classes and instances.

# Object-oriented Programming

The evolution of programming has taken us from a sequence of step-by-step instructions in a single flow of control to a more organized approach whereby blocks of code could be cordoned off into named subroutines and defined functionality. Structured or procedural programming lets us organize our programs into logical blocks, often repeated or reused. Creating applications becomes a more logical process; actions are chosen which meet the specifications, then data is created to be subjected to those actions. Deitel & Deitel refer to structured programming as "action-oriented" due to the fact that logic must be "enacted" on data which has no associated behaviors.

However, what if we *could* impose behavior on data? What if we were able to create or program a piece of data modeled after real-life entities which embodies both data characteristics along with behaviors? If we were then able to access the data attributes via a set of defined interfaces (a.k.a. a set of accessor functions), such as an automated teller machine (ATM) card or a personal check to access your bank account, then we would have a system of "objects" where each could interact not only with itself, but also with other objects in a larger picture.

Object-oriented programming takes this evolutionary step by enhancing structured programming to enable a data/behavior relationship: Data and logic are now described by a single abstraction with which to create these objects. Real-world problems and entities are stripped down to their bare essentials, providing that abstraction from which they can be coded or implemented into objects and be able to interact with other objects in the system which models and hopefully solves these problems. Classes provide the definitions of such objects, and instances are realizations of such definitions. Both are vital components for object-oriented design (OOD), which simply means to build your system architected in an object-oriented fashion.

## Relationship between OOD and OOP

Object-oriented design does not specifically require an object-oriented programming language. Indeed, OOD can be performed in purely structural languages such as C, but this requires more effort on the part of the programmer who must build data types with object qualities and characteristics. Naturally, OOP is simplified when a language has built-in OO properties that enable smoother and more rapid development of OO programs.

Conversely, an object-oriented language does not necessarily force one to write OO programs. C++ can be used simply as a "better C." As you are no doubt aware, neither classes nor OOP are required for everyday Python programming. Even though it is a language which is object-oriented by design and which has constructs to support OOP, Python does not restrict nor require you to write OO code for your application. Rather, OOP is a powerful tool which is at your disposal when you are ready to evolve, learn, transition, or otherwise move towards OOP. The creator of Python often refers to this phenomena as being able to "see the forest through the trees."

## Real-world Problems

One of the most important reasons to consider working in OOD is that it provides a direct approach to modeling and solving real-world problems and situations. For example, let us attempt to model an automobile mechanic shop where you would take your car in for repair. There are two general entities we would have to create: humans who interact with and in such a "system," and a physical location for the activities which define a mechanic shop. Since there are more of and different types of the former, we will describe them first, then conclude with the latter.

A class called `Person` would be created to represent all humans involved in such an activity. Instances of `Person` would include the `Customer,` the `Mechanic,` and perhaps the `Cashier.` Each of these instances would have similar as well as unique behaviors. For example, all would have the `talk()` method as a means of vocal communication as well as a `drive_car()` method. Only the `Mechanic` would have the `repair_car()` method and only the `Cashier` would have a `ring_sale()` method. The `Mechanic` will have a `repair_certification` attribute while all `Persons` would have a `drivers_license` attribute.

Finally, all of these instances would be participants in one overseeing class, called the `RepairShop,` which would have `operating_hours,` a data attribute which accesses time functionality to determine when `Customers` can bring in their vehicles and when `Employees` such as `Mechanics` and Cashiers show up for work. The `RepairShop` might also have a `AutoBay` class which would have instances such as `SmogZone,` `TireBrakeZone,` and perhaps one called `GeneralRepair.`

The point of our fictitious `RepairShop` is to show one example of how classes and instances plus their behaviors can be used to model a true-to-life scenario. You can probably also imagine classes such as an `Airport,` a `Restaurant,` a `ChipFabPlant,` a `Hospital,` or even a `MailOrderMusic` business, all complete with their own participants and functionality.

## *Buzzword-compliance

For those of you who are already familiar with all the lingo associated with OOP, here is how Python stacks up:

### Abstraction/Implementation

*Abstraction* refers to the modeling of essential aspects, behavior, and characteristics of real-world problems and entities, providing a relevant subset as the definition of a programmatic structure which can realize such models. Abstractions not only contain the data attributes of such a model, but also define interfaces with that data. An *implementation* of such an abstraction is the realization of that data and the interfaces which go along with it. Such a realization should remain hidden from and irrelevant to

the client programmer. Class objects in Python provide the ability to create such abstractions, and implementation details are left to the designer.

### Encapsulation/Interfaces

*Encapsulation* describes the concept of data/information hiding and providing *interfaces* or accessor functions to the data attributes. Direct access to data by any client, bypassing the interfaces, goes against the principles of encapsulation, but the programmer is free to allow such access. As part of the implementation, the client should not even know how the data attributes are architected within the abstraction. In Python, all class attributes are public, but names may be "mangled" to discourage unauthorized access, but otherwise not prevented. It is up to the designer to provide the appropriate interfaces to the data so that the client programmer does not have to resort to manipulating the encapsulated data attributes.

### Composition

*Composition* extends our description of classes, enabling multiple yet distinct classes to be combined into a larger entity to solve a real-world problem. Composition describes a singular, complex system such as a class made up of other, smaller components such as other classes, data attributes, and behaviors, all of which are combined, embodying "has-a" relationships. For example, the `RepairShop` "has a" `Mechanic` (hopefully at least one) and also "has a" `Customer` (again, hopefully at least one).

These components are composed either via *association,* meaning that access to subcomponents is granted (for the `RepairShop,` a customer may enter and request a `SmogCheck,` the client programmer interfacing with components of the `RepairShop`), or *aggregation,* encapsulating components which are then accessed only via defined interfaces, and again, hidden from the client programmer. Continuing our example, the client programmer may be able to make a `SmogCheck` request on behalf of the `Customer,` but has no ability to interact with the `SmogZone` part of the `RepairShop` which is accessed only via internal controls of the `RepairShop` when the `smogCheckCar()` method is called. Both forms of composition are supported in Python.

### Derivation/Inheritance/Hierarchy

*Derivation* describes the creation of subclasses, new classes which retain all desired data and behavior of the existing class type but permit modification or other customization, all without having to modify the original class definition. *Inheritance* describes the means by which attributes of a subclass are "bequeathed from" an ancestor class. From our earlier example, a `Mechanic` may have more car skill attributes than a `Customer,` but individually, each "is a" `Person,` so it is valid to invoke the `talk()` method, which is common to all instances of `Person,` for either of them. *Hierarchy* describes multiple "generations" of derivation which can be depicted graphically as a "family tree," with successive subclasses having relationships with ancestor classes.

### Generalization/Specialization

*Generalization* describes all the traits a subclass has with its parent and ancestor classes, so subclasses are considered to have an "is-a" relationship with ancestor classes because a derived object (instance) is an "example" of an ancestor class. For example, a `Mechanic` "is a" `Person`, a `Car` "is a" `Vehicle`, etc. In the family tree diagram we alluded to above, we can draw lines from subclasses to ancestors indicating "is-a" relationships. *Specialization* is the term which describes all the customization of a subclass, i.e., what attributes which make it differ from its ancestor classes.

### Polymorphism

The concept of *polymorphism* describes how objects can be manipulated and accessed using attributes and behaviors they have in common without regard to their specific class. Polymorphism indicates the presence of dynamic (a.k.a. late, run-time) binding, allowing for overloading and run-time type determination and verification. Many OO languages use "signatures" to determine which version of an overloaded method to call, but since Python calls are universal or generic without type determination, overloading is unnecessary and is not supported in the language.

### Introspection/Reflection

*Introspection* is what gives you, the programmer, the ability to perform an activity such as "manual type checking." Also called *reflection,* this property describes how information about a particular object can be accessed by itself during run-time. Would it not be great to have the ability to take an object passed to you and be able to find out what it is capable of? This is a powerful feature which you will encounter frequently in this chapter. The `dir()` and `type()` built-in functions would have a very difficult time working if Python did not support for some sort of introspection capability. Keep an eye out for these calls as well as for special attributes like `__dict__`, `__name__`, `__doc__`, `__members__`, and `__methods__`. You may even be familiar with some of them already!

**NOTE**

In other object-oriented programming languages, the term "object" may refer specifically to class instances, even more so when all data types of those languages are classes. Not so with Python. Because data types in Python are not classes, not all objects are, therefore, class instances.

Some languages also consider defining a class to be synonymous with creating a new type. Again, this is not the case with Python, but it is similar. Python has a fixed number of predefined types and these remain constant. (Creating a new type in Python is a non-trivial task, requiring implementation as an extension, and it is out of the scope of this text.) When creating classes, you can give them behavior characteristics of types but they are not considered types.

However, Python is an object-oriented programming language and considers all entities generically as objects because they do share some common semantics, yet are still

distinct enough to be different types of objects. In summary, classes, instances, and types are not related to each other (with the exception that a class defines an object which is realized as an instance, another type of object).

Bottom line: (all) classes are class objects, (all) instances are instance objects, neither are types, and everything is an object. Also see the Core Note in *Section 13.5.1*.

## Classes

Recall that a class is a data structure that we can use to define objects which hold together data values and behavioral characteristics. Classes are entities which are the programmatic form of an abstraction for a real-world problem, and instances are realizations of such objects. One analogy is to liken classes to blueprints or molds with which to make real objects (instances). So why the term "class?" The term most likely originates from using classes to identify and categorize biological families of species to which specific creatures belong and can be derived into similar yet distinct subclasses. Many of these features apply to the concept of classes in programming.

In Python, class declarations are very similar to function declarations, a header line with the appropriate keyword followed by a suite as its definition, as indicated below:

```
def functionName(args):
    'function documentation string'
    function_suite

class                                                ClassName:
    'class documentation string"
    class_suite
```

The fact that such a declaration is "larger" than a standard type declaration should be proof that classes in Python are much more than standard types. A class is like a Python container type on steroids. Not only can it hold multiple data items but it can also support its own set of functions, which we have seen before, called methods. You may be asking what other advantages classes have over standard container types such as lists and dictionaries.

Standard types are fixed, cannot be customized, and come with a hard-coded set of attributes. Data types also do not provide individual namespaces for objects nor can they be used to derive "sub-types." Objects contained in lists are unrelated except for the name of their container. Its members are accessed only via an index offset into an array-like data structure. All lists have the same set of methods. The same goes for dictionaries, which also have a common set of methods and provide key access to their members (who are also unrelated except for their container name).

In this section, we will take a close look at classes and what types of attributes they have. Just remember to keep in mind that even though classes are objects (everything in Python is an object), they are not realizations of the objects they are defining. We will look at instances in the next chapter, so stay tuned for that. For now, the limelight is strictly beamed on class objects.

When you create a class, you are practically creating your own kind of data entity. All instances of that class are similar, but classes differ from each other (and so will instances of different classes by nature). Rather than playing with toys that came from the manufacturer and were bestowed upon you as gifts, why not design and build your own toys to play with?

### Creating Classes

Python classes are created using the **`class`** keyword. In the simple form of class declarations, the name of the class immediately follows the keyword:

```
class ClassName:
     'class documentation string'
     class_suite
```

*`class_suite`* consists of all the component statements, defining class members, data attributes, and functions. Classes are generally defined at the top-level of a module so that instances of a class can be created anywhere in a piece of source code where the class is defined.

### Declaration vs. Definition

As with Python functions, there is no distinction between declaring and defining classes because they occur simultaneously, i.e., the definition (the class suite) immediately follows the declaration (header line with the **`class`** keyword) and the always recommended, but optional, documentation string. Likewise, all methods must also be defined at this time. If you are familiar with the OOP terms, Python does not support *pure virtual functions* (à la C++) or *abstract methods* (as in Java), which coerce the programmer to define a method in a subclass.

## Class Attributes

What is an attribute? An attribute is a data or functional element which belongs to another object and is accessed via the familiar dotted-attribute notation. Some Python types such as complex numbers have data attributes (`real` and `imag`), while others such as lists and dictionaries have methods (functional attributes).

One interesting side note about attributes is that when you are accessing an attribute, it is also an object and may have attributes of its own which you can then access, leading to a

chain of attributes, i.e., `myThing.subThing.subSubThing`, etc. Some familiar examples are:

- `sys.stdout.write('foo')`
- `print myModule.myClass.__doc__`
- `myList.extend(map(upper, open('x').readlines()))`

Class attributes are tied only to the classes in which they are defined, and since instance objects are the most commonly used objects in everyday OOP, instance data attributes are the primary data attributes you will be using. Class data attributes are useful only when a more "static" data type is required which is independent of any instances, hence the reason we are making the next section advanced, optional reading.

In the succeeding subsection, we will briefly describe how methods in Python are implemented and invoked. In general, all methods in Python have the same restriction: They require an instance before they can be called.

## *Class Data Attributes

Data attributes are simply variables of the class we are defining. They can be used like any other variable in that they are set when the class is created and can be updated either by methods within the class or elsewhere in the main part of the program.

Such attributes are better known to OO programmers as *static members, class variables,* or *static data.* They represent data that is tied to the class object they belong to and are independent of any class instances. If you are a Java or C++ programmer, this type of data is the same as placing the `static` keyword in front of a variable declaration.

Static members are generally used only to track values associated with classes. In most circumstances, you would be using instance attributes rather than class attributes. We will compare the differences between class and instance attributes when we formally introduce instances.

Here is an example of using a class data attribute (`foo`):

```
>>> class C:
…         foo = 100
>>> print C.foo
0
>>> C.foo = C.foo + 1
>>> print C.foo
101
```

Note that nowhere in the code above do you see any references to class instances.

## Methods

A method, such as the `myNoActionMethod` method of the `MyClass` class in the example below, is simply a function defined as part of a class definition (thus making methods class attributes). This means that `myMethod` applies only to objects (instances) of `MyClass` type. Note how `myNoActionMethod` is tied to its instance because invocation requires both names in the dotted attribute notation:

```
>>> class MyClass:
        def myNoActionMethod(self):
            pass

>>> myInstance = MyClass()
>>> myInstance.myNoActionMethod()
```

Any call to `myNoActionMethod` by itself as a function fails:

```
>>> myNoActionMethod()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    myNoActionMethod()
NameError: myNoActionMethod
```

A `NameError` exception is raised because there is no such function in the global namespace. The point is to show you that `myNoactionMethod` is a method, meaning that it belongs to the class and is not a name in the global namespace. If `myNoActionMethod` was defined as a function at the top-level, then our call would have succeeded.

We show you below that even calling the method with the class object fails.

```
>>> MyClass.myNoActionMethod()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    MyClass.myNoActionMethod()
TypeError: unbound method must be called with class
instance 1st argument
```

This `TypeError` exception may seem perplexing at first because you know that the method is an attribute of the class and so are wondering why there is a failure. We will explain this next.

### Binding (Bound and Unbound Methods)

In keeping with OOP tradition, Python imposes the restriction that methods cannot be invoked without instances. An instance must be used to perform method calls. This restriction describes Python's concept of binding, where methods must be bound (to an instance) in order to be invoked directly. Unbound methods may also be called, but an instance object must be provided explicitly in order for the invocation to succeed. However, regardless of binding, methods are inherently attributes of the class they are defined in, even if they are almost always invoked via an instance. We will further explore bound and unbound methods later in Section 13.7.

### Static Methods

Python does not support *static methods* (a.k.a. *static member functions*), functions which are associated only with a class and not with any particular instances. They are either functions which help manage static class data or are global functions which have some sort of functionality related to the class they are defined in. Since Python does not support static methods, a standard global function is the workaround when static method functionality is desired. More details on how this is accomplished can be found in Section 13.7.2.

## Determining Class Attributes

There are two ways to determine what attributes a class has. The simplest way is to use the `dir()` built-in function. An alternative is to access the class dictionary attribute `__dict__`, one of a number of special attributes that is common to all classes. Let us take a look at an example:

```
>>> class MyClass:
…         'MyClass class definition'
…         myVersion = '1.1'         # static data
…         def showMyVersion(self):  # method
…             print MyClass.myVersion
…
```

Using the class defined above, let us use `dir()` and the special class attribute `__dict__` to see this class's attributes:

```
>>> dir(MyClass)
['__doc__', '__module__', 'showMyVersion', 'myVersion']
>>>
>>> MyClass.__dict__
{'__doc__': None, 'myVersion': 1, 'showMyVersion':
<function showMyVersion at 950ed0>, '__module__':
'__main__'}
```

As you can tell, `dir()` returns a list of an object's attributes while `__dict__` is a dictionary, with the attribute names as keys and whose values are the data values of the corresponding attributes.

The output also reveals two familiar attributes of our class `MyClass`, `showMyVersion` and `myVersion`, as well as a couple of new ones. These attributes, `__doc__` and `__module__`, are special class attributes which all classes have (in addition to `__dict__`). The `vars()` built-in function returns the contents of a class's `__dict__` attribute when passed the class object as its argument.

## Special Class Attributes

For any class `C`, Table 13.1 represents a list of all the special attributes of `C`:

**Table 13.1. Special Class Attributes**

| | |
|---|---|
| *C*.__name__ | string name of class *C* |
| *C*.__doc__ | documentation string for class *C* |
| *C*.__bases__ | tuple of class *C*'s parent classes |
| *C*.__dict__ | attributes of *C* |
| *C*.__module__ | module where *C* is defined (new in 1.5) |

Using the class `MyClass` we just defined above, we have the following:

```
>>> MyClass.__name__
'MyClass'
>>> MyClass.__doc__
'MyClass class definition'
>>> MyClass.__bases__
()
>>> MyClass.__dict__
{'__doc__': None, 'myVersion': 1, 'showMyVersion':
<function showMyVersion at 950ed0>, '__module__':
'__main__'}
>>> MyClass.__module__
'__main__'
```

__name__ is the string name for a given class. This may come in handy in cases where a string is desired rather than a class object. Even some built-in types have this attribute, and we will use one of them to showcase the usefulness of the __name__ string.

The type object is an example of one built-in type that has a __name__ attribute. Recall that type() returns a type object when invoked. There may be cases where we just want the string indicating the type rather than an object. We can use the __name__ attribute of the type object to obtain the string name. Here is an example:

```
>>> stype = type('What is your quest?')
>>> stype                          # stype is a type object
<type 'string'>
>>> stype.__name__                 # get type as a string
'string'
>>>
>>> type(3.14159265)               # also a type object
<type 'float'>
>>> type(3.14159265).__name__      # get type as a string
'float'
```

__doc__ is the documentation string for the class, similar to the documentation string for functions and modules, and must be the first unassigned string succeeding the header line. The documentation string is *not* inherited by derived classes, as an indication that they must contain their own documentation strings.

__bases__ deals with inheritance which we will cover later in this chapter; it contains a tuple which consists of a class's parent classes.

The aforementioned __dict__ attribute consists of a dictionary containing the data attributes of a class. When accessing a class attribute, this dictionary is searched for the attribute in question. If it is not found in __dict__, the hunt continues in the dictionary of base classes, in "depth-first search" order. The set of base classes is searched in sequential order, left-to-right in the same order as they are defined as parent classes in a class declaration. Modification of a class attribute affects only the current class's dictionary; no base class __dict__ attributes are ever modified.

Python supports class inheritance across modules, so to better clarify a class's description, the __module__ was introduced in version 1.5 so that a class name is fully qualified with its module. We present the following example:

```
>>> class C:
…       pass
…
>>> C
<class __main__.C at 81201f0>
>>> C.__module__
'__main__'
```

The fully-qualified name of class `C` is `"__main__.C"`, i.e. *`source_module.class_name`*. If class `C` was located in an imported module, such as `mymod`, we would see the following:

```
>>> from mymod import C
>>> C                           # class C in Python 1.5.2
<class mymod.C at 8120be0>
>>> C.__module__
'mymod'
```

In previous versions of Python without the special attribute `__module__`, it was much more difficult to ascertain the location of a class simply because classes did not use their fully-qualified names. For example, if we were to perform the same module import and access the class, you can see that no source module name for class `C` is available:

```
>>> from mymod import C
>>> C                        # class C in Python 1.4
<class C at 8120be0>
```

## Instances

Whereas a class is a data structure definition type, an instance is a declaration of a variable of that type. In other words, instances are classes brought to life. Once a blueprint is provided, the next step to bring them to fruition. Instances are the objects which are used primarily during execution, and all instances are of type "instance."

### Instantiation: Creating Instances by Invoking Class Object

Most other OO languages provide a `new` keyword with which to create an instance of a class. Python's approach is much simpler. Once a class has been defined, creating an instance is no more difficult that calling a function—literally. Instantiation is realized with use of the function operator, as in the following example:

```
>>> class MyClass:              # define class
…         pass
>>> myInstance = MyClass()      # instantiate class
>>> type(MyClass)               # class is of class type
<type 'class'>
>>> type(myInstance)            # instance is of
instance type
<type 'instance'>
```

**NOTE**

The use of the term "type" in Python may perhaps differ from the general connotation of an instance being of the type of class it was created from. In *Chapter 4*, we introduced all Python objects as data entities with three characteristics: an ID, a type, and a value. An object's type dictates the behavioral properties of such objects in the Python system, and these types are a subset of all types which Python supports.

User-defined "types" such as classes are categorized in the same manner. Classes share the same type, but have different IDs and values (their class definitions). The fact that all classes are defined with the same syntax, that they can be instantiated, and that all have the same core properties leads to the conclusion that they have common characteristics which allow them to fall under the same category. Classes are unique objects which differ only in definition, hence they are all the same "type" in Python. Class instances follow the same argument.

Do not let Python's nomenclature fool you; instances are most assuredly related to the class they were instantiated from and would not have any other relationship to other instances (unless they were of a subclass or base class).

To avoid confusion, keep the following in mind: When you are defining a class, you are not creating a new type. You are just defining a unique class type, but it is still a class. When you instantiate classes, the resulting object is always an instance. Even though instances may be instantiated from different classes, they are still (generically) class instances.

As you can see, creating instance `myInstance` of class `MyClass` consists of "calling" the class: `MyClass()`. The return object of the call is an instance object. We also verified the data types of `MyClass` and `myInstance` using `type(): MyClass` is a class object and `myInstance` is an instance object. To take this even further, we tell you now that *all* classes are of the same type (type class), and *all* instances are of the same type (type instance). (You can verify this using the `type()` built-in function.)

## `__init__()` Constructor Method

When the class is invoked, the first step in the instantiation process is to create the instance object. Once the object is available, a check to see if a constructor has been implemented is called. By default, no special actions are enacted on the instance without the overriding of the constructor, the special method `__init__()`. Any special action desired requires the programmer to implement `__init__()`, overriding its default behavior. If `__init__()` has not been implemented, the object is then returned and the instantiation process is complete.

However, if `__init__()` *has* been implemented, then that special method is invoked and the instance object passed in as the first argument (`self`), just like a standard method call. Any arguments passed to the class invocation call are passed on to `__init__()`. You can practically envision the call to create the class as a call to the constructor.

`__init__()` is one of many special methods which can be defined for classes. Some of these special methods are predefined with inaction as their default behavior (such as `__init__()`) and must be overridden for customization while others should be implemented on an as-needed basis. We will go over special methods in Section 13.13 later on in this chapter.

## `__del__()` Destructor Method

Likewise, there is an equivalent *destructor* special method called `__del__()`. However, due to the way Python manages garbage collection of objects (by reference counting), this function is not executed until all references to an instance object have been removed. Destructors in Python are methods which provide special processing before instances are deallocated and are not commonly implemented since instances are seldom deallocated explicitly.

**NOTE**

Python does not provide any internal mechanism to track how many instances of a class have been created nor to keep tabs on what they are. You can explicitly add some code to the class definition and perhaps `__init__()` and `__del__()` if such functionality is desired. The best way is to keep track of the number of instances using a static member. It would be dangerous to keep track of instance objects by saving references to them, because you must manage these references properly or else your instances will never be deallocated (because of your extra reference to them)! An example follows:

```
                                                          class
myClass:
   count = 0                # use static data for count
   def __init__(self):      # constructor, incr. count
          myClass.count = myClass.count + 1
   def __del__(self):       # destructor, decr. count
          myClass.count = myClass.count - 1
          assert myClass > 0  # cannot have < 0 instances
   def howMany(self):       # return count
          return myClass.count
>>> a = myClass()
>>> b = myClass()
>>> b.howMany()
2
>>> a.howMany()
2
>>> del b
>>> a.howMany()
```

```
1
>>> del a
>>> myClass.count
0
```

In the following example, we create (and override) both the `__init__()` and `__del__()` constructor and destructor functions, respectively, then instantiate the class and assign more aliases to the same object. The `id()` built-in function is then used to confirm that all three aliases reference the same object. The final step is to remove all the aliases by using the `del` statement and discovering when and how many times the destructor is called.

```
>>> class C:                    # class declaration
        def __init__(self):     # constructor
            print 'initialized'
        def __del__(self):      # destructor
            print 'deleted'

>>> c1 = C()                    # instantiation
initialized
>>> c2 = c1                     # create additional alias
>>> c3 = c1                     # create a third alias
>>> id(c1), id(c2), id(c3)      # all refer to same object
(11938912, 11938912, 11938912)
>>> del c1                      # remove one reference
>>> del c2                      # remove another reference
>>> del c3                      # remove final reference
deleted                         # destructor finally invoked
```

Notice how, in the above example, the destructor was not called until all references to the instance of class `C` were removed, i.e., when the reference count has decreased to zero. If for some reason your `__del__()` method is not being called when you are expecting it to be invoked, this means that somehow your instance object's reference count is not zero, and there may be some other reference to it that you are not aware of that is keeping your object around.

Also note that the destructor is called exactly once, the first time the reference count goes to zero and the object deallocated. This makes sense because any object in the system is allocated and deallocated only once.

## Instance Attributes

Instances have only data attributes (methods are strictly class attributes) and are simply data values which you want to be associated with a particular instance of any class and

are accessible via the familiar dotted-attribute notation. These values are independent of any other instance or of the class it was instantiated from. When an instance is deallocated, so are its attributes.

## "Instantiating" Instance Attributes (or, Creating a Better Constructor)

Instance attributes can be set any time after an instance has been created, in any piece of code that has access to the instance. However, one of the key places where such attributes are set is in the constructor, `__init__()`.

### Constructor First Place to Set Instance Attributes

The constructor is the earliest place that instance attributes can be set because `__init__()` is the first method called after instance objects have been created. There is no earlier opportunity to set instance attributes. Once `__init__()` has finished execution, the instance object is returned, completing the instantiation process.

### Default Arguments Provide Default Instance Setup

One can also use `__init__()` along with default arguments to provide an effective way in preparing an instance for use in the real world. In many situations, the default values represent the most common cases for setting up instance attributes, and such use of default values precludes them from having to be given explicitly to the constructor. We also outlined some of the general benefits of default arguments in Section 11.5.2.

Example 13.1 shows how we can use the default constructor behavior to help us calculate some sample total room costs for lodging at hotels in some of America's large metropolitan areas.

The main purpose of our code is to help someone figure out the daily hotel room rate, including any state sales and room taxes. The default is for the general area around San Francisco, which has an 8.5% sales tax and a 10% room tax. The daily room rate has no default value, thus it is required for any instance to be created.

The setup work is done after instantiation by `__init__()` in lines 4–8, and the other core part of our code is the `calcTotal()` method, lines 10–14. The job of `__init__()` is to set the values needed to determine the total base room rate of a hotel room (not counting room service, phone calls, or other incidental items). `calcTotal()` is then used to either determine the total daily rate or for an entire stay if the number of days is provided. The `round()` built-in function is used to round the calculation to the closest penny (two decimal places). Here is some sample usage of this class:

**Example 13.1. Using Default Arguments with Instantiation (`hotel.py`)**

Class definition for a fictitious hotel room rate calculator. The `__init__()` constructor method initializes several instance attributes. A `calcTotal()` method is used to determine either a total daily room rate, or the total room cost for an entire stay.

```
 <$nopage>
001 1  class HotelRoomCalc:
002 2      'Hotel room rate calculator'
003 3
004 4      def __init__(self, rt, sales=0.085, rm=0.1):
005 5          '''HotelRoomCalc default arguments:
006 6          sales tax == 8.5% and room tax == 10%'''
007 7          self.salesTax = sales
008 8          self.roomTax = rm
009 9          self.roomRate = rt
010 10
011 11     def calcTotal(self, days=1):
012 12         'Calculate total; default to daily rate'
013 13         daily = round((self.roomRate * \
014 14             (1 + self.roomTax + self.salesTax)), 2)
015 15         return float(days) * daily
016  <$nopage>

>>> sfo = HotelRoomCalc(299)                    # new instance
>>> sfo.calcTotal()                             # daily rate
354.32
>>> sfo.calcTotal(2)                            # 2-day rate
708.64
>>> sea = HotelRoomCalc(189, 0.086, 0.058)      # new instance
>>> sea.calcTotal()
216.22
>>> sea.calcTotal(4)
864.88
>>> wasWkDay = HotelRoomCalc(169, 0.045, 0.02)  # new instance
>>> wasWkEnd = HotelRoomCalc(119, 0.045, 0.02)  # new instance
>>> wasWkDay.calcTotal(5) + wasWkEnd.calcTotal() # 7-day rate
1026.69
```

The first two hypothetical examples were San Francisco, which used the defaults, and then Seattle, where we provided different sales tax and room tax rates. The final example, Washington, D.C., extended the general usage by calculating a hypothetical longer stay: a five-day weekday stay plus a special rate for one weekend day, assuming a Sunday departure to return home.

Do not forget that all the flexibility you get with functions, such as default arguments, apply to methods as well. The use of variable-length arguments is another good feature to use with instantiation (based on an application's needs, of course).

### Constructor Should Return `None`

As you are now aware, invoking a class object with the function operator creates a class instance, which is the object returned on such an invocation, as in the following example:

```
>>> class MyClass:
…         pass
>>> myInstance = MyClass()
>>> myInstance
<__main__.MyClass instance at 95d390>
```

If a constructor is defined, it should not return any object because the instance object is automatically returned after the instantiation call. Correspondingly, `__init__()` should not return any object (or return `None`); otherwise, there is a conflict of interest because only the instance should be returned. Attempting to return any object other than None will result in a `TypeError` exception:

```
>>> class MyClass:
…     def __init__(self):
…         print 'initialized'
…         return 1
…
>>> myInstance = MyClass()
initialized
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    myInstance = MyClass()
TypeError: __init__() should return None
```

## Determining Instance Attributes

The `dir()` built-in function can be used to show all instance attributes in the same manner that it can reveal class attributes:

```
>>> class C:
…         pass
>>> c = C()
>>> c.foo = 'roger'
>>> c.bar = 'shrubber'
>>> dir(c)
['bar', 'foo']
```

Similar to classes, instances also have a `__dict__` special attribute (also accessible by calling `vars()` and passing it an instance), which is a dictionary representing its attributes:

```
>>> c.__dict__
{'foo': 'roger', 'bar': 'shrubber'}
```

## Special Instance Attributes

Instances have only two special attributes (see Table 13.2). For any instance *I*:

| Table 13.2. Special Instance Attributes | |
|---|---|
| *I*.__class__ | class from which *I* is instantiated |
| *I*.__dict__ | attributes of *I* |

We will now take a look at these special instance attributes using the class C and its instance c:

```
>>> class C:      # define class
…     pass
…
>>> c = C()                    # create instance
>>> dir(c)                     # instance has no attributes
[]
>>> c.__dict__                 # yep, definitely no attributes
{}
>>> c.__class__                # class that instantiated us
<class __main__.C at 948230>
```

As you can see, c currently has no data attributes; but we can add some and recheck the __dict__ attribute to make sure they have been added properly:

```
>>> c.foo = 1
>>> c.bar = 'SPAM'
>>> '%d can of %s please' % (c.foo, c.bar)
'1 can of SPAM please'
>>> dir(c)
['bar', 'foo']
>>> c.__dict__
{'foo': 1, 'bar': 'SPAM'}
```

The __dict__ attribute consists of a dictionary containing the attributes of an instance. The keys are the attribute names, and the values are the attributes' corresponding data values. You will only find instance attributes in this dictionary—no class attributes nor special attributes.

**NOTE**

Although the __dict__ attributes for both classes and instances are mutable, it is recommended that you not modify these dictionaries unless or until you know exactly what you are doing. Such modification contaminates your OOP and may have unexpected side effects. It is more acceptable to access and manipulate attributes using the familiar dotted-attribute notation. One of the few cases where you would modify the __dict__ attribute directly is when you are overriding the __setattr__ special method. Implementing __setattr__() is another adventure story on its own, full of traps and pitfalls such as infinite recursion and corrupted instance objects—but that is another tale for another time.

## Built-in Type Attributes

Built-in types also have attributes, and although they are technically not class instance attributes, they are sufficiently similar to get a brief mention here. Type attributes do not have an attribute dictionary like classes and instances (__dict__), so how do we figure out what attributes built-in types have? The convention for built-in types is to use two special attributes, __methods__ and __members__, to outline any methods and/or data attributes. Complex numbers are one example of a built-in type with both methods and attributes, so we will use its __methods__ and __members__ to help us hunt down its attributes:

```
>>> aComplex = (1+2j)     # create a complex number
>>> type(aComplex)        # display its type
<type 'complex'>
>>> aComplex.__members__ # reveal its data attributes
['imag', 'real']
>>> aComplex.__methods__ # reveal its methods
['conjugate']
```

Now that we know what kind of attributes a complex number has, we can access the data attributes and call its methods:

```
>>> aComplex.imag
2.0
>>> aComplex.real
1.0
>>> aComplex.conjugate()
(1-2j)
```

Attempting to access __dict__ will fail because that attribute does not exist for built-in types:

```
>>> aComplex.__dict__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: __dict__
```

Our final remark for this section is to note that the `__members__` and `__methods__` special attributes is simply a convention. New types defined in external or third-party extension modules may not choose to implement them, although it is highly recommended.

## Instance Attributes vs. Class Attributes

We first described class data attributes in Section 13.4.1. As a brief reminder, class attributes are simply data values associated with a class and not any particular instances like instance attributes are. Such values are also referred to as static members because their values stay constant, even if a class is invoked due to instantiation multiple times. No matter what, static members maintain their values independent of instances unless explicitly changed. Comparing instance attributes to class attributes is almost like the comparison between automatic and static variables, if you are familiar with these concepts from other languages.

There are a few aspects of class attributes versus instance attributes that should be brought to light. The first is that you can access a class attribute with either the class or an instance, provided that the instance does not have an attribute with the same name.

### Access to Class Attributes

Class attributes can be accessed via a class or an instance. In the example below, when class `C` is created with the version class attribute, naturally access is allowed using the class object, i.e., `C.version`. When instance `c` is created, Python provides a default, read-only instance attribute which is an alias to the class attribute, i.e., `c.version`:

```
>>> class C:                          # define class
…       version = 1.0                 # static member
…
>>> c = C()                           # instantiation
>>> C.version                         # access via class
1.0
>>> c.version                         # access via instance
1.0
>>> C.version = C.version + .1        # update (only) via class
>>> C.version                         # class access
1.1
>>> c.version                         # instance access, which
1.1                                   # also reflected change
```

However, access via to a class attribute via an instance attribute is strictly read-only (see below for what happens if you try to update one), so we can only update the value when referring to it using the class, as in the `C.version` increment statement above. Attempting to set or update the class attribute using the instance name is not allowed and will create an instance attribute.

### Assignment Creates Local Instance Attribute

Any type of assignment of a local attribute will result in the creation and assignment of an instance attribute, just like a regular Python variable. If a class attribute exists with the same name, it is overridden in the instance:

```
>>> dir(C)
['__doc__', '__module__', 'version']
>>> dir(c)
[]
>>> c.version = 100      # attempt to update class attr
>>> c.version
100
>>> C.version            # nope, class attr unchanged
1.1
>>> dir(c)               # confirm new instance attr created
['version']
```

In the above code snippet, a new instance attribute named `version` is created, overriding the reference to the class attribute. However, the class attribute itself is unscathed and still exists in the class domain and can still be accessed as a class attribute, as we can see above.

To confirm that a new instance attribute was added, the call to `dir()` in the above code snippet reveals no attributes for instance `c`, while class `C` had three attributes (`__doc__`, `__module__`, and `version`). Calling `dir()` again on `c` after the assignment yields one new attribute, `version`.

What would happen if we delete this new reference? To find out, we will use the `del` statement on `c.version`.

```
>>> del c.version        # delete instance attribute
>>> dir(c)
[]
>>> c.version            # can now access class attr again
1.1
```

Now let us try to update the class attribute again, but this time, we will just try an innocent increment:

```
>>> c.version = c.version + 1.0
>>> c.version
2.1
>>> dir(c)
['version']
>>> C.version
1.1
```

It is still a "no go." We again created a new instance attribute while leaving the original class attribute intact. The expression on the right-hand side of the assignment evaluates the original class variable, adds 1.0 to it, and assigns it to a newly-created instance attribute. Note that the following is an equivalent assignment, but it may perhaps provide more clarification:

```
c.static = C.static + 1.0
```

### Class Attributes More Persistent

Static members, true to their name, hang around while instances (and their attributes) come and go (hence independent of instances). Also, if a new instance is created after a class attribute has been modified, the updated value will be reflected:

```
>>> class C:
…       spam = 100           # class attribute
…
>>> c1 = C()                  # create an instance
>>> c1.spam                   # access class attr thru inst.
100
>>> C.spam = C.spam + 100     # update class attribute
>>> C.spam                    # see change in attribute
200
>>> c1.spam                   # confirm change in attribute
200
>>> c2 = C()                  # create another instance
>>> c2.spam                   # verify class attribute
200
>>> del c1                    # remove one instance
>>> C.spam = C.spam + 200     # update class attribute again
>>> c2.spam                   # verify that attribute changed
400
```

## Binding and Method Invocation

Now we need to readdress the Python concept of binding, which is associated only with method invocation. We will first review the facts about methods. First, a method is simply a function defined as part of a class. This means that methods are class attributes

(not instance attributes). Second, methods can be invoked only when there is an instance of the class in which the method was defined. When there is an instance present, the method is considered bound. Without an instance, a method is considered unbound. And third, the first argument in any method definition is the variable `self`, which represents the instance object which invokes the method.

**NOTE**

The variable `self` is used in class instance methods to reference the instance which the method is bound to. Because a method's instance is always passed as the first argument in any method call, `self` is the name that was chosen to represent the instance. You are required to put `self` in the method declaration (you may have noticed this already) but do not need to actually use the instance (`self`) within the method.

If you do not use `self` in your method, you might consider creating a regular function instead, unless you have a particular reason not to. After all, your code, because it does not use the instance object in any way, "unlinks" its functionality from the class, making it seem more like a general function.

We will now create a class C with a method called `showSelf()` which will display more information about the instance object that was just created:

```
>>> class C:
…         def showSelf(self):
…             self
…             type(self)
…             id(self)

>>> c = C()
>>> c.showSelf()
<__main__.C instance at 94abe0>
<type 'instance'>
9743328
```

Now let us take a look directly at the instance to see if the information matches—and it does:

```
>>> c
<__main__.C instance at 94abe0>
>>> type(c)
<type 'instance'>
>>> id(c)
9743328
```

In other object-oriented languages, `self` is named `this`.

## Invoking Bound Methods

Methods, bound or not, is made up of the same code. The only difference is whether there is an instance present so that the method can be invoked. Recall that even though `self` is required as the first argument in every method declaration, it never needs to be passed explicitly when you invoke it from an instance. The interpreter automatically performs that task for you.

Once again, here is an example of invoking a bound method, first found in Section 13.4.2:

```
>>> class MyClass:
…         def myNoActionMethod(self):
…             pass
…
>>>
>>> myInstance = MyClass()              # create instance
>>> myInstance.myNoActionMethod()       # invoke method
```

To invoke a method, use the name of the instance and the name of the method in dotted attribute notation followed by the function operator and any arguments.

Also in Section 13.4.2, we briefly noted the failure of invoking the method with the class name. The cause of this failure is that no instance was given to the method. Without invoking a method using an instance (and having that instance passed automatically as `self` to the method), the interpreter complained that `self` was not passed in. What we did wrong was to invoke an unbound method without an instance.

## Invoking Unbound Methods

There are generally two reasons a programmer might attempt to invoke an unbound method. One is when a programmer is trying to implement static methods (which is not supported in Python), and the other is when a specific instance of the class defining the method is not available. We begin our tale by describing a workaround to the lack of static method support in Python.

### Static Method Workaround

Static methods are generally desired in two different scenarios. The first is a situation where a programmer wants to keep his or her global or local namespace "pure," by not adding another function to the corresponding namespace. Or secondly, perhaps it is a small or insignificant function which is somehow related to the class he wants to define it in, or maybe it is a function that helps manage static data. The first case presents a relatively weak argument, but there is some merit for the latter case, this static member

management function. Here, we simply want to invoke the method in a functional sense (meaning independent of instances) in order to update static data.

```
>>> class C:
…         version = 1.0          # static data attribute
…         def updateVersion(self, newv):
…             C.version = newv   # update static data
…
```

And of course, invoking this method without an instance gives us the same `TypeError` exception we have seen before:

```
>>> C.updateVersion(2.0)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    C.updateVersion()
TypeError: unbound method must be called with class
instance 1st argument
```

The only workaround, which may be unpleasant, is to give up and move the function to the global domain which has access to class `C` (and thus the attributes of `C`). We can obtain the desired functionality:

```
>>> def updateVersion(newv):
…         C.version = newv
…
>>> updateVersion(2.0)
>>> C.version
2.0
```

One problem with using a global function as the solution is that you do not get the "feel" that `updateVersion()` is a class method, because it is not. The desire is to call `C.updateVersion()` or something. Yes, there are other, more sinister workarounds which new Python programmers should avoid because, to paraphrase from the Python FAQ, "[if] you don't understand why you'd ever want to do this, that's because you are pure of mind, and you probably never will want to do it! This is dangerous trickery, [and] not recommended when avoidable."

### Convenient Instance Unavailable

When a bound method is invoked, for example, `instance.method(x, y)`, the interpreter would be executing the equivalent of `method(instance, x, y)`, as in the following example:

```
>>> class MyData:
…         def myMethod(self, arg):
…             print 'called myMethod with:', arg
…
>>>
>> myInstance = MyData()
>>> myInstance.myMethod('grail')
called myMethod with: grail
```

The call `myInstance.myMethod('grail')` results in the equivalent call of `myMethod(myInstance, 'grail')`, which matches the function signature, `myMethod(self, arg)`.

However, invoking an unbound method does not work quite as well. Since there is no bound instance, the method call will fail because the call would be simply `method(x, y)` rather than `method(instance, x, y)`. Here is the error one more time:

```
>>> MyData.myMethod(932)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    MyData.myMethod(932)
TypeError: unbound method must be called with class
instance 1st argument
```

Notice that the `TypeError` exception states, "unbound method must be called with class instance 1st argument." Ah, that's the problem. The first argument passed to the method was an integer, not a class instance. The function signatures did not match, i.e., `myMethod(932)` versus `myMethod(self, arg)`. What if we *did* pass an instance as the first argument (to make the signatures match)?

```
>>> MyData.myMethod(myInstance, 932)
called myMethod with: 932
```

Presto! It now works. So an interesting result of having an instance is that you can now invoke an unbound method if you explicitly provide the instance so that the call is `method(instance, x, y)`.

This is all rather nice, but what if there is no instance nearby for us to use? What if in the above example, we had not created the `myInstance` object? Well, then we would *not* have been able to invoke `myMethod()` then. Does this ever happen? The answer is yes and comes into play in Section 13.9 below. Such situations require the invocation of a base class method from the method of a derived class. We've already seen it once, in the "Creating a Subclass" subsection of Section 13.1.

## Composition

Once a class is defined, the goal is to use it as a model programmatically, embedding this object throughout your code, intermixing use with other data types and the logical flow of execution. There are two ways of utilizing classes in your code. The first is composition. This is where different classes are mingled with and into other classes for added functionality and code reusability. You may perhaps create instances of your class inside a larger class, containing other attributes and methods enhancing the use of the original class object. The other way is with derivation and discussed in the next section.

For example, let us imagine an enhanced design of the address book class we created at the beginning of the chapter. If, during the course of our design, we created separate classes for names, addresses, etc., we would want to integrate that work into our `AddrBookEntry` class, rather than have to redesign each of those supporting classes. We have the added advantages of time and effort saved, as well as more consistent code—when bugs are fixed in that same piece of code, that change is reflected in all the applications which reuse that code.

Such as a class would perhaps contain `Name` and `Phone` instances, not to mention others like `StreetAddress, Phone` (home, work, telefacsimile, pager, mobile, etc.), `Email` (home, work, etc.), and possibly a few `Date` instances (birthday, wedding, anniversary, etc.). Here is a simple example with some of the classes mentioned above:

```
class NewAddrBookEntry:          # class definition
  'new address book entry class'
  def __init__(self, nm, ph):    # define constructor
     self.name = Name(nm)        # create Name instance
     self.phone = Phone(ph)      # create Phone instance
     print 'Created instance for:', self.name
```

The `NewAddrBookEntry` class is a composition of itself and other classes. This defines a "has-a" relationship between a class and other classes it is composed of. For example, our `NewAddrBookEntry` class "has a" `Name` class instance and a `Phone` instance, too.

Creating composite objects enables such additional functionality and make sense because the classes have nothing in common. Each class manages its own namespace and behavior. When there are more intimate relationships between objects, a more elegant solution is the concept of derivation.

## Subclassing and Derivation

Composition works fine when classes are distinct and are a required component of larger classes, but when you desire "the same class but with some tweaking," derivation is a more logical option.

One of the more powerful aspects of OOP is the ability to take an already-defined class and extend it or make modifications to it without affecting other pieces of code in the system that use the currently-existing classes. OOD allows for class features to be "inherited" by "descendant" classes or "subclasses." These subclasses "derive" the core of their attributes from "base" (a.k.a. ancestor, super) classes. In addition, this derivation may be extended for multiple generations. Classes involved in a one-level derivation (or are adjacent vertically in a class tree diagram) have a "parent" and "child" class relationship. Those classes which derive from the same parent (or are adjacent horizontally in a class tree diagram) have a "sibling" relationship. Parent and all higher-level classes are considered ancestors.

Using our example from the previous section, let us imagine having to create different types of address books. We are talking about more than just creating multiple instances of address books—in this case, all objects have everything in common. What if we wanted a `BusinessAddressBook` class whose entries would contain more work-related attributes such as job position, phone number, and e-mail address? This would differ from a `PersonalAddressBook` class which would contain more family-oriented information such as home address, relationship, birthday, etc.

For both of these cases, we do not want to design these classes from scratch, because it would duplicate the work already accomplished to create the generic `AddressBook class`. Wouldn't it be nice to subsume all the features and characteristics of the `AddressBook class` and add specialized customization for your new, yet related, classes? This is the entire motivation and desire for class derivation.

## Creating Subclasses

As we have seen earlier, the general syntax for declaring a base class looks like this:

```
class                                                    ClassName:
     'optional class documentation string'
     class_suite
```

Derived classes are declared much like their parent class; however, a list of base classes to inherit from are given after the class name:

```
class                                                    SubClassName
(ParentClass1[, ParentClass2, …]):
     'optional class documentation string'
     class_suite
```

We have already seen some examples of classes and subclasses so far, but here is another simple example:

```
>>> class Parent:                  # define parent class
…       def parentMethod(self):
…           print 'calling parent method'

>>> p = Parent()                   # instance of parent
>>> dir(Parent)                    # parent class attributes
['__doc__', '__module__', 'parentMethod']
>>> p.parentMethod()
calling parent method
>>>
>>> class Child(Parent):           # define child class
…       def childMethod(self):
…           print 'calling child method'

>>> c = Child()                    # instance of child
>>> dir(Child)                     # child class attributes
['__doc__', '__module__', 'childMethod']
>>> c.childMethod()                # child calls its method
calling child method
>>> c.parentMethod()               # calls parent's method
calling parent method
```

## Inheritance

*Inheritance* describes how the attributes of base classes are "bequeathed" to a derived class. A subclass inherits attributes of any of its base classes whether they be data attributes or methods.

We present an example below. `P` is a simple class with no attributes. `C` is a class with no attributes which derives from (and therefore is a subclass of) `P`:

```
>>> class P:                       # parent class
…       pass
>>> class C(P):                    # child class
…       pass
>>>
>>> c = C()                        # instantiate child
>>> c.__class__                    # child "is a" parent
<class __main__.C at 8120c98>
>>> C.__bases__                    # child's parent class(es)
(<class __main__.P at 811fc98>,)
```

Because `P` has no attributes, nothing was inherited by `C`. Let us make our example more useful by giving `P` some attributes:

```
>>> class P:                    # parent class
…       'P class'
…         def __init__(self):
…             print 'created an instance of', \
…                 self.__class__.__name__
…
>>> class C(P):                 # child class
…         pass
```

We now create `P` with a documentation string (`__doc__`) and a constructor which will execute when we instantiate `P`, as in this interactive session:

```
>>> p = P()                     # parent instance
created an instance of P
>>> p.__class__                 # class that created us
<class __main__.P at 811f900>
>>> P.__bases__                 # parent's parent class(es)
()
>>> P.__doc__                   # parent's doc string
'P class'
>>> dir(P)                      # parent class attributes
['__doc__', '__init__', '__module__']
```

The "created an instance" output comes directly from `__init__()`. We also display some more about the parent class P for your information. Since `P` is not a subclass, its `__bases__` attribute is empty. We will now instantiate `C`, showing you how the `__init__()` (constructor) method is inherited with its execution:

```
>>> c = C()                             # child instance
created an instance of C
>>> c.__class__                         # class that created us
<class __main__.C at 812c1b0>
>>> C.__bases__                         # child's parent class(es)
(<class __main__.P at 811f900>,)
>>> C.__doc__                           # child's doc string
>>>
>>> dir(C)                              # child class attributes
['__doc__', '__module__']
```

`C` has no declared method `__init__()`, yet there is still output when instance `c` of class `C` is created. The reason is that `C` inherits `__init__()` from `P`. The `__bases__` tuple now lists `P` as its parent class.

You will notice that some special data attributes are not inherited, the most notable of which is `__doc__`. Each class should have its own documentation string. It does not

make sense inheriting special class attributes because the values generally relate to one specific class.

## __bases__ Class Attribute

We briefly introduced the __bases__ class attribute in Section 13.4.4, which is a tuple containing the set of parent classes for any (sub)class. Note that we specifically state "parents" as opposed to all base classes (which includes all ancestor classes). Classes which are not derived will have an empty __bases__ attribute. Let us look at an example of how to make use of __bases__.

```
>>> class A: pass                  # define class A
…
>>> class B(A): pass               # subclass of A
…
>>> class C(B): pass               # subclass of B (and indirectly, A)
…
>>> class D(A,B): pass             # subclass of A and B
…
>>> C.__bases__
(<class __main__.B at 8120c90>,)
>>> D.__bases__
(<class __main__.A at 811fc90>, <class __main__.B at 8120c90>)
```

In the example above, although C is a derived class of both A (through B) and B, C's parent is B, as indicated in its declaration, so only B will show up in C.__bases__. On the other hand, D inherits from two classes, A and B. (Multiple inheritance is covered in Section 13.10.4.)

## Overriding Methods through Inheritance

Let us create another function in P that we will override in its child class:

```
>>> class P:
…         def foo(self):
…             print 'Hi, I am P-foo()'
…
>>> p = P()
>>> p.foo()
Hi, I am P-foo()
```

Now let us create the child class C, subclassed from parent P:

```
>> class C(P):
…         def foo(self):
```

```
…             print 'Hi, I am C-foo()'
…
>>> c = C()
>>> c.foo()
Hi, I am C-foo()
```

Although `C` inherits `P`'s `foo()` method, it is overridden because `C` defines its own `foo()` method. One reason for overriding methods is because you may want special or different functionality in your subclass. Your next obvious question then must be, "Can I call a base class method which I overrode in my subclass?"

The answer is yes, but this is where you will have to invoke an unbound base class method, explicitly providing the instance of the subclass, as we do here:

```
>>> P.foo(c)
Hi, I am P-foo()
```

Notice that we already had an instance of `P` called `p` from above, but that is nowhere to be found in this example. We do not need an instance of `P` to call a method of `P` because we have an instance of a *subclass* of `P` which we can use, `c`.

**NOTE**

When deriving a class with a constructor `__init__()`, if you do not override `__init__()`, it will be inherited and automatically invoked. But if you do override `__init__()` in a subclass, the base class `__init__()` method is not invoked automatically when the subclass is instantiated.

```
>>> class P:
…         def __init__(self):
…             print "calling P\xd5 s constructor"
…
>>> class C(P):
…         def __init__(self):
…             print "calling C\xd5 s constructor"
…
>>> c = C()
created an instance of C
```

If you want the base class `__init__()` invoked, you need to do that explicitly in the same manner as we just described, calling the base class (unbound) method with an instance of the subclass. Updating our class `C` appropriately results in the following desired execution:

```
>>> class C(P):
…         def __init__(self):
…             P.__init__(self)
…             print "calling C's constructor"
…
>>> c = C()
calling P's constructor
calling C's constructor
```

In the above example, we call the base class `__init__()` method before the rest of the code in our own `__init__()` method. It is fairly common practice (if not mandatory) to initialize base classes for setup purposes, then proceed with any local setup. This rule makes sense because you want the inherited object properly initialized and "ready" by the time the code for the derived class constructor runs, because it may require or set inherited attributes.

Those of you familiar with C++ would call base class constructors in a derived class constructor declaration by appending a colon to the declaration followed by calls to any base class constructors. Java programmers have no choice—base class constructors must always be called as the first thing that happens in derived class constructors. Python's use of the base class name to invoke a base class method is directly comparable to Java's when using the keyword `super.`

## Deriving Standard Types

One limitation is that Python types are *not* classes, meaning that we cannot derive subclasses from them. Not all is lost though, because of the many different special default attribute methods we can implement to emulate the standard types (see the Core Note in Section 4.2 as well as Sections 6.14.2 and 13.12).

## Multiple Inheritance

Python allows for subclassing from multiple base classes. This feature is commonly known as "multiple inheritance." Python supports a limited form of multiple inheritance whereby a depth-first searching algorithm is employed to collect the attributes to assign to the derived class. Unlike other Python algorithms which override names as they are found, multiple inheritance takes the first name that is found.

Our example below consists of a pair of parent classes, a pair of children classes, and one grandchild class.

```
class P1:                           # parent class 1
   def foo(self):
```

```
      print 'called P1-foo()'

class P2:                           # parent class 2
   def foo(self):
      print 'called P2-foo()'
   def bar(self):
      print 'called P2-bar()'

class C1(P1,P2):                    # child 1 der. from P1, P2
   pass

class C2(P1,P2):                    # child 2 der. from P1, P2
   def foo(self):
      print 'called C2-foo()'
   def bar(self):
      print 'called C2-bar()'

class GC(C1,C2):                    # define grandchild class
   pass                             # derived from C1 and C2
```

Upon executing the above declarations in the interactive interpreter, we can confirm that only the first attributes encountered are used.

```
>> gc = GC()
>>> gc.foo()        # GC ? C1 ? P1
called P1-foo()
>>> gc.bar()        # GC ? C1 ? P1 ? P2
called P2-bar()
```

Again, you can always call a specific method by invoking the method using its fully-qualified name and providing a valid instance:

```
>>> C2.foo(gc)
called C2-foo()
```

## Built-in Functions for Classes, Instances, and Other Objects

### issubclass()

The `issubclass()` Boolean function determines if one class is a subclass or descendant of another class. It has the following syntax:

```
issubclass(sub, sup)
```

`issubclass()` returns 1 if the given subclass *`sub`* is indeed a subclass of the superclass *`sup`*. This function allows for an "improper" subclass, meaning that a class is viewed as a subclass of itself, so the function returns 1 if `sub` is either the same class as `sup` or derived from `sup`. (A "proper" subclass is strictly a derived subclass of a class.)

If we were to implement the `issubclass()` built-in function ourselves, it may look something like the following:

```
def my_issubclass(sub, sup):
    if sub is sup or sup in sub.__bases__:
        return 1
    for cls in sub.__bases__:
        if my_issubclass(cls, sup):
            return 1
        else:
            return 0
    return 0
```

We first check to see if they are both the same class. Since we allow for improper subclasses, then we would indicate a successful inquiry if both classes are the same. We also return 1 if `sup` is a parent class of `sub`. This check is accomplished by looking at the `__bases__` attribute of sub.

If both of those tests fail, then we need to start moving up the family tree to see if `sup` is an ancestor of `sub`. This is accomplished by checking the parent classes of `sub` to see if they are subclasses of `sup`. If that fails, then we check the grandparent classes, and so on, moving up the tree to see if any of those classes are subclasses of `sup`. This is a depth-first recursive check where all ancestors are checked. If all return a negative result, we return 0 for failure.

### isinstance()

The `isinstance()` Boolean function is useful for determining if an object is an instance of a given class and has the following syntax:

```
isinstance(obj1, obj2)
```

`isinstance()` returns 1 if *`obj1`* is an instance of class *`obj2`* or is an instance of a subclass of *`obj2`*, as indicated in the following examples:

```
>>> class C1: pass
…
>>> class C2: pass
…
>>> c1 = C1()
```

```
>>> c2 = C2()
>>> isinstance(c1, C1)
1
>>> isinstance(c2, C1)
0
>>> isinstance(c1, C2)
0
>>> isinstance(c2, C2)
1
>>> isinstance(C2, c2)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    isinstance(C2, c2)
TypeError: second argument must be a class
```

Note that the second argument should be a class; otherwise, you get a `TypeError`. The only exception is if the second argument is a type object. This is allowed because you can also use `isinstance()` to check if an object *`obj1`* is of the type *`obj2`*, i.e.,

```
>>> isinstance(4, type(4))
1
>>> isinstance(4, type(''))
0
>>> isinstance('4', type(''))
1
```

If we were to implement the `isinstance()` built-in function ourselves, it may look something like the following:

```
def my_isinstance(obj1, obj2):
  if obj2 is type(type(0)):       # check if obj2 is type obj
    return type(obj1) is obj2
  if obj1.__class__ is obj2:      # check if obj1 inst of obj2
    return 1
return my_issubclass(obj1.__class__, obj2)
```

`isinstance()` may appear simpler than `issubclass()`, but you will notice that we use `issubclass()`. Without it, we would have to reimplement `issubclass()`, so in actuality, `isinstance()` is a bit longer than `issubclass()`.

Our rendition of `isinstance()` works like this: We first check to see if we are dealing with objects and types by confirming whether `obj2` is a type object. If so, then we perform the check and return the result. Otherwise, we are dealing with classes and instances, so the test proceeds to check if instance `obj1` is actually a real instance of class `obj2`. If it is, then we are done. Otherwise, we recursively check to see if the class of

which `obj1` is an instance is a descendant of `obj2`. If it is, then we return 1 for yes and 0 otherwise.

### Proxy for Missing Functionality

Both `is*()` built-in functions (`issubclass()` and `isinstance()`) are new to Python as of version 1.5. Prior to 1.5, you would have to implement them yourself, as we did above, as well as create your own routine which proxies for the missing functions. While new functions are not always part of Python updates, it is quite possible that due to uncontrollable forces, you are required to use older versions of Python which do not support the new functionality. The solution would be to implement your own solutions and integrate them into your code so that your system at least *behaves* like a more recent version of the interpreter.

We present one possible solution below, which attempts to import the `__builtin__` module (hopefully loading `issubclass()` and `isinstance()`), but if not, creates aliases for our homebrewed functions which reference the code we implemented above:

```
if '__builtins__' not in dir() and __import__ in dir():
     __builtins__ = __import__('__builtin__')

if 'issubclass' not in dir(__builtins__):
     issubclass = my_issubclass

if 'isinstance' not in dir(__builtins__):
     isinstance = my_isinstance
```

We will now invoke these functions on the classes and instance we defined in the previous section. Recall that the `P*` classes are parent classes, the `C*` classes are child classes, and the `GC` class is the "grandchild" class.

```
>>> for eachCls in (P1, P2, C1, C2, GC):
…       print "is GC subclass of", eachCls.__name__, '?', \
…               issubclass(GC, eachCls)
…       print "is 'gc' an instance of", eachCls, '?', \
…               isinstance(gc, eachCls)

is GC subclass of P1 ? 1
is 'gc' an instance of __main__.P1 ? 1
is GC subclass of P2 ? 1
is 'gc' an instance of __main__.P2 ? 1
is GC subclass of C1 ? 1
is 'gc' an instance of __main__.C1 ? 1
is GC subclass of C2 ? 1
is 'gc' an instance of __main__.C2 ? 1
is GC subclass of GC ? 1
is 'gc' an instance of __main__.GC ? 1
```

### hasattr(), getattr(), setattr(), delattr()

The *attr() functions can work with all kinds of objects, not just classes and instances. However, since they are most often used with those objects, we present them here.

The hasattr(obj, attr) function is Boolean and its only function is to determine whether or not an object has a particular attribute, presumably used as a check before actually *trying* to access that attribute. The getattr() and setattr() functions retrieve and assign values to object attributes, respectively. getattr() will raise an AttributeError exception if you attempt to read an object that does not have the requested attribute. setattr() will either add a new attribute to the object or replace a pre-existing one. The delattr() function removes an attribute from an object.

Here are some examples using all the *attr() BIFs:

```
>>> class myClass:
…       def __init__(self):
…           self.foo = 100
…
>>> myInst = myClass()
>>> dir(myInst)
['foo']
>>> hasattr(myInst, 'foo')
1
>>> getattr(myInst, 'foo')
100
>>> hasattr(myInst, 'bar')
0
>>> setattr(myInst, 'bar', 'my attr')
>>> dir(myInst)
['bar', 'foo']
>>> getattr(myInst, 'bar')
'my attr'
>>> delattr(myInst, 'foo')
>>> dir(myInst)
['bar']
>>> hasattr(myInst, 'foo')
0
```

### dir()

We first experienced dir() in Exercises 2-12, 2-13, and 4-7. In those exercises, we used dir() to give us information about all the attributes of a module. We now know that dir() can be applied to any other objects with attributes—these include classes, class instances, files, lists, complex numbers, and so on. As long as an object has a __dict__ attribute dictionary, and/or the __members__ and __methods__ lists, dir() will work.

Built-in types do not have a `__dict__` attribute dictionary and rely primarily on `__members__` and `__methods__` as a convention:

```
>>> dir(3+3j)              # complex number attributes
['conjugate', 'imag', 'real']
>>>
>>> (3+3j).__dict__
Traceback (innermost last):
File "<stdin>", line 1, in ?
AttributeError: __dict__
>>>
>>> (3+3j).__members__
['imag', 'real']
>>>
>>> (3+3j).__methods__
['conjugate']
>>>
>>> f = open('/etc/motd')
>>> dir(f)            # file oject attributes
['close', 'closed', 'fileno', 'flush', 'isatty', 'mode',
'name', 'read', 'readinto', 'readline', 'readlines',
'seek', 'softspace', 'tell', 'truncate', 'write',
'writelines]
>>> f.__dict__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: __dict__
>>> f.__members__
['closed', 'mode', 'name', 'softspace']
>>> f.__methods__
['close', 'fileno', 'flush', 'isatty', 'read',
'readinto', 'readline', 'readlines', 'seek', 'tell',
'truncate', 'write', 'writelines]
>>> f.close()
```

### vars()

The `vars()` built-in function is similar to `dir()` except that any object given as the argument must have a `__dict__` attribute. `vars()` will return a dictionary of the attributes (keys) and values of the given object based on the values in its `__dict__` attribute. If the object provided does not have such an attribute, an `TypeError` exception is raised. If no object is provided as an argument to `vars()`, it will display the dictionary of attributes (keys) and the values of the local namespace, i.e., `locals()`. We present below an example of calling `vars()` with a class instance:

```
>>> class C:
…       pass

>>> c=C()
>>> c.foo = 100
```

```
>>> c.bar = 'Python'
>>> c.__dict__
{'foo': 100, 'bar': 'Python'}
>>> vars(c)
{'foo': 100, 'bar': 'Python'}
```

Table 13.3 summarizes the built-in functions for classes and class instances.

**Table 13.3. Built-in Functions for Classes, Instances, and Other Objects**

| Built-in Function | Description |
|---|---|
| `issubclass(sub, sup)` | returns 1 if class *sub* is a subclass of class *sup*, 0 otherwise |
| `isinstance(obj1, obj2)` | returns 1 if instance *obj1* is an instance of class *obj2* or is an instance of a subclass of *obj2*; will also return 1 if *obj1* is of type *obj2* |
| `hasattr(obj, attr)` | returns 1 if *obj* has attribute *attr* (given as a string) |
| `getattr(obj, attr)` | retrieves attribute *attr* of *obj*; same as return *obj.attr*; `AttributeError` exception raised if `attr` is not an attribute of *obj* |
| `setattr(obj, attr, val)` | sets attribute *attr* of *obj* to value `val`, overriding any previously-existing attribute value, otherwise, attribute is created; same as *obj.attr = val* |
| `delattr(obj, attr)` | removes attribute *attr* (given as a string) from *obj*; same as **del** *obj.attr* |
| `dir(obj=None)` | returns a list of the attributes of *obj*; if *obj* not given, `dir()` displays local namespace attributes, i.e., `locals().keys()` |
| `vars(obj=None)` | returns a dictionary of the attributes and values of *obj*; if *obj* not given, `vars()` displays local namespace dictionary (attributes and values), i.e., `locals()` |

## Type vs. Classes/Instances

Unlike languages such as Java, Python standard types are not classes and variables are not instances of such classes. Rather, they are simply primitive types that do not allow for direct derivation. (As it turns out, a class is simply a specific type of primitive built-in that *does* allow for derivation.)

Python supports a fixed number of built-in types; thus, even instances themselves are of a singular type ("instance"), and classes are all of type "class." (Also review the Core Note earlier in this chapter which posed the issue of why instances are all of the same type.)

Subclassing of standard types is not possible, though often desired. Python provides for some elegant solutions, one of which is to create a new class which has the behavior of a standard type. This allows for the most flexibility because you are in control of your new type at all times. The other solution allows for the use of a pre-existing type by "wrapping" a standard type in a class. (We will address wrapping in Section 13.15). By "wrapping," we mean provide the standard type as the data object of the class and provide accessor methods which allow for the same type of functionality. This is also a perfect mechanism for designing and developing a custom data type for an application, which will be our focus for the upcoming section.

## Customizing Classes with Special Methods

We covered two important aspects of methods in preceding sections of this chapter, the first being that methods must be bound (to an instance of their corresponding class) before they can be invoked. The other important matter is that there are two special methods which provide the functionality of constructors and destructors, namely `__init__()` and `__del__()` respectively.

In fact, `__init__()` and `__del__()` are part of a set of special methods which can be implemented. Some have the predefined default behavior of inaction while others do not and should be implemented where needed. These special methods allow for a powerful form of extending classes in Python. In particular, they allow for:

Emulating standard types

Overloading operators

Special methods enable classes to emulate standard types by overloading standard operators such as `+`, `*`, and even the slicing subscript and mapping operator `[ ]`. As with most other special reserved identifiers, these methods begin and end with a double underscore ( `__` ). Table 13.4 presents a list of all special methods and their descriptions.

**Table 13.4. Special Methods for Customizing Classes**

| Special Method | Description |
|---|---|

| Core | |
|---|---|
| `C.__init__(self[, arg1, …])` | constructor (with any optional arguments) |
| `C.__del__(self)` | destructor |
| `C.__repr__(self)` | evaluatable string representation; `repr()` built-in and " operator |
| `C.__str__(self)` | printable string representation; `str()` built-in and **`print`** statement |
| `C.__cmp__(self, obj)` | object comparison; `cmp()` built-in |
| `C.__call__(self, *args)` | denote callable instances |
| `C.__nonzero__(self)` | define false value for object |
| `C.__len__(self)` | "length" (appropriate for class); `len()` built-in |
| Attributes | |
| `C.__getattr__(self, attr)` | get attribute; `getattr()` built-in |
| `C.__setattr__(self, attr, val)` | set attribute; `setattr()` built-in |
| `C.__delattr__(self, attr)` | delete attribute; **`del`** statement |
| Customizing Classes / Emulating Types | |

| **Numeric Types: binary operators**[a] | |
|---|---|
| `C.__*add__(self, obj)` | addition; `+` operator |
| `C.__*sub__(self, obj)` | subtraction; `-` operator |
| `C.__*mul__(self, obj)` | multiplication; `*` operator |
| `C.__*div__(self, obj)` | division; `/` operator |
| `C.__*mod__(self, obj)` | modulo/remainder; `%` operator |
| `C.__*divmod__(self, obj)` | division and modulo; `divmod()` built-in |
| `C.__*pow__(self, obj[, mod])` | exponentiation; `pow()` built-in; ** operator |
| `C.__*lshift__(self, obj)` | left shift; `<<` operator |
| `C.__*rshift__(self, obj)` | right shift; `>>` operator |
| `C.__*and__(self, obj)` | bitwise AND; `&` operator |
| `C.__*or__(self, obj)` | bitwise OR; `\|` operator |
| `C.__*xor__(self, obj)` | bitwise XOR; `^` operator |
| Numeric Types: unary operators | |
| `C.__neg__(self)` | unary negation |

| | |
|---|---|
| `C.__pos__(self)` | unary no-change |
| `C.__abs__(self)` | absolute value; `abs()` built-in |
| `C.__invert__(self)` | bit inversion; `~` operator |
| Numeric Types: numeric conversion | |
| `C.__complex__(self, com)` | convert to complex; `complex()` built-in |
| `C.__int__(self)` | convert to int; `int()` built-in |
| `C.__long__(self)` | convert to long; `long()` built-in |
| `C.__float__(self)` | convert to float; `float()` built-in |
| Numeric Types: base representation (string) | |
| `C.__oct__(self)` | octal representation; `oct()` built-in |
| `C.__hex__(self)` | hexadecimal representation; `hex()` built-in |
| Numeric Types: numeric coercion | |
| `C.__coerce__(self, num)` | coerce to same numeric type; `coerce()` built-in |
| **Sequence Types**[a] | |
| `C.__len__(self)` | number of items in sequence |

| | |
|---|---|
| C.__getitem__(self, ind) | get single sequence element |
| C.__setitem__(self, ind, val) | set single sequence element |
| C.__delitem__(self, ind) | delete single sequence element |
| C.__getslice__(self, ind1, ind2) | get sequence slice |
| C.__setslice__(self, i1, i2, val) | set sequence slice |
| C.__delslice__(self, ind1, ind2) | delete sequence slice |
| C.__contains__(self, val)[b] | test sequence membership; **in** keyword |
| C.__*add__(self, obj) | concatenation; + operator |
| C.__*mul__(self, obj) | repetition; * operator |
| Mapping Types | |
| C.__len__(self) | number of items in mapping |
| C.__hash__(self) | hash function value |
| C.__getitem__(self, key) | get value with given key |
| C.__setitem__(self, key, val) | set value with given key |
| C.__delitem__(self, key) | delete value with given key |

[a] "*" either nothing (self OP obj), "r" (obj OP self), or (new in 2.0) "i" for in-place operation, i.e., __add__, __radd__, or __iadd__

[b] New in 1.6

The "Core" group of special methods denotes the basic set of special methods which can be implemented without emulation of any specific types. The "Attributes" group helps manage instance attributes of your class. The "Numeric Types" set of special methods can be used to emulate various numeric operations, including those of the standard (unary and binary) operators, conversion, base representation, and coercion. There are also special methods to emulate sequence and mapping types. Implementation of some of these special methods will overload operators so that they work with instances of your class type.

Numeric binary operators in the table annotated with a wildcard asterisk in their names are so denoted to indicate that there are multiple versions of those methods with slight differences in their name. The asterisk either symbolizes no additional character in the string, or a single "r" to indicate a right-hand-side operation. Without the "r," the operation occurs for cases which are of the format `self OP obj;` the presence of the "r" indicates the format `obj OP self.` For example, `__add__(self, obj)` is called for `self + obj,` and `__radd__(self, obj)` would be invoked for `obj + self.`

Augmented assignment, new in Python 2.0, introduces the notion of "in-place" operations. An "i" in place of the asterisk implies a combination left-hand side operation plus an assignment, as in `self = self OP obj.` For example, `__iadd__(self, obj)` is called for `self = self + obj.`

## Simple Class Customization Example (`oPair`)

For our first example, let us create a simple class consisting of an ordered pair `(x, y)` of numbers. We will represent this data in our class as a 2-tuple. In the code snippet below, we define the class with a constructor that takes a pair of values and stores them as the data attribute of our `oPair` class:

```
class oPair:                            # ordered pair
    def __init__(self, obj1, obj2):    # constructor
        self.data = (obj1, obj2)        # assign attribute
```

Using this class, we can instantiate our objects:

```
>>> myPair = oPair(6, -4)               # create instance
>>> myPair                              # calls repr()
<oPair instance at 92bb50>
>>> print myPair                       # calls str()
<oPair instance at 92bb50>
```

Unfortunately, neither **print** (using `str()`) nor the actual object's string representation (using `repr()`) reveals much about our object. One good idea would be to implement either `__str__()` or `__repr__()`, or both so that we can "see" what our object looks like. In other words, when you want to display your object, you actually want to see something meaningful rather than the generic Python object string (`<object type at id>`). We want to see an ordered pair (tuple) with the current data values in our object. Without further ado, let us implement `__str__()` so that the ordered pair is displayed:

```
def __str__(self):            # str() string representation
    return str(self.data)     # convert tuple to string

__repr__ = __str__            # repr() string representation
```

Since we also want to use the same piece of code for `__repr__()`, rather than copying the code verbatim, we use our sense of code reusability and simply create an alias to `__str__()`. Now our output has been greatly improved:

```
>>> myPair = oPair(-5, 9)# create instance
>>> myPair               # repr() calls __repr__()
(-5, 9)
>>> print myPair         # str() calls __str__()
(-5, 9)
```

What is the next step? Let us say we want our objects to interact. For example, we can define the addition operation of two `oPair` objects, `(x1, y1)` and `(x2, y2)`, to be the sum of each individual component. Therefore, the "sum" of two oPair objects is defined as a new object with the values `(x1 + x2, y1 + y2)`. We implement the `__add__()` special method in such a way that we calculate the individual sums first, then call the class constructor to return a new object. Finally, we alias `__add__` as `__radd__` since the order does not matter—in other words, numeric addition is commutative. The definitions of `__add__` and `__radd__` are featured below:

```
def __add__(self, other):       # add two oPair objects
    return self.__class__(self.data[0] + other.data[0],
                          self.data[1] + other.data[1])
```

The new object is created by invoking the class as in any normal situation. The only difference is that from within the class, you typically would not invoke the class name directly. Rather, you take `__class__` attribute of `self` which is the class from which

`self` was instantiated and invoke *that.* Because `self.__class__` is the same as `oPair`, calling `self.__class__()` is the same as calling `oPair()`.

Now we can perform additions with our newly-overloaded operators. Reloading our updated module, we create a pair of `oPair` objects and "add" them, producing the sum you see below:

```
>>> pair1 = oPair(6, -4)
>>> pair2 = oPair(-5, 9)
>>> pair1 + pair2
(1, 5)
```

A `TypeError` exception occurs when attempting to use an operator for which the corresponding special method(s) has(have) not been implemented yet:

```
>>> pair1 * pair2
Traceback (innermost last):
  File "<stdin", line 1, in ?
    pair1 * pair2
TypeError: __mul__ nor __rmul__ defined for these operands
```

Obviously, our result would have been similar if we had not implemented `__add__` and `__radd__`. The final example is related to existing data which we may want to use. Let us say that we have some 2-tuples floating around in our system, and in order to create `oPair` objects with them currently, we would have to split them up into individual components to instantiate an `oPair` object:

```
aTuple = (-3, -1)
pair3 = oPair(aTuple[0], aTuple[1])
```

But rather than splitting up the tuple and creating our objects as in the above, wouldn't it be nice if we could just feed this tuple into our constructor so that it can handle it there? The answer is yes, but not by overloading the constructor as the case may be with other object-oriented programming languages. Python does not support overloading of callables, so the only way to work around this problem is to perform some manual introspection with the `type()` built-in function.

In our update to `__init__()` below, we add an initial check to see if what we have is a tuple. If it is, then we just assign it directly to the data attribute. Otherwise, this would mean a "regular" instantiation, meaning that we expect a pair of numbers to be passed.

```
def __init__(self, obj1, obj2=None):# constructor
  if type(obj1) == type(()):        # tuple type
      self.data = obj1
  else:
    if obj2 == None:                # part of values
      raise TypeError, \
            'oPair() requires tuple or numeric pair'
    self.data = (obj1, obj2)
```

Note in the above code that we needed to give a default value of `None` to `obj2`. This allows only one object to be passed in if it is a tuple. What we do *not* want is to allow only the creation of an `oPair` type without a second value, hence our additional check to see if `obj2` is `None` in the **`else`** clause. We can now make our call in a more straightforward manner:

```
aTuple = (-3, -1)
pair3 = oPair(aTuple)
>>> pair3
(-3, -1)
>>> pair3 + pair1
(3, -5)
```

Hopefully, you now have a better understanding of operator overloading, why you would want to do it, and how you can implement special methods to accomplish that task. If you are interested in a more complex customization, continue with the optional section below.

## *More Complex Class Customization Example (`NumStr`)

Let us create another new class, `NumStr`, consisting of a number-string ordered pair, called *n* and *s,* respectively, using integers as our number type. Although the "proper" notation of an ordered pair is `(n, s)`, we choose to represent our pair as `[n :: s]` just to be different. Regardless of the notation, these two data elements are inseparable as far as our model is concerned. We want to set up our new class, called `NumStr`, with the following characteristics:

### Initialization

The class should be initialized with both the number and string; if either (or both) is missing, then 0 and the empty string should be used, i.e., `n=0` and `s=''`, as defaults.

### Addition

We define the addition operator functionality as adding the numbers together and concatenating the strings; the tricky part is that the strings must be concatenated in the correct order. For example, let `NumStr1 = [n1 :: s1]` and `NumStr2 = [n2 :: s2]`.

Then `NumStr1 + NumStr2` is performed as `[n1 + n2 :: s1 + s2]` where `+` represents addition for numbers and concatenation for strings.

### Multiplication

Similarly, we define the multiplication operator functionality as multiplying the numbers together and repeating or concatenating the strings, i.e., `NumStr1 * NumStr2 = [n1 * n2 :: s1 * s2].`

### False Value

This entity has a false value when the number has a numeric value of zero and the string is empty, i.e., when `NumStr = [0 :: ''].`

### Comparisons

Comparing a pair of `NumStr` objects, i.e., `[n1 :: s1]` vs. `[n2 :: s2],` we find 9 different combinations (i.e., `n1 > n2` and `s1 < s2, n1 == n2` and `s1 > s2,` etc.) We use the normal numeric and lexicographic compares for numbers and strings, respectively, i.e., the ordinary comparison of `cmp(obj1, obj2)` will return an integer less than zero if `obj1 < obj2,` greater than zero if `obj1 > obj2,` or equal to zero if the objects have the same value.

The solution for our class is to add both of these values and return the result. The interesting thing is that `cmp()` does not like to return values other than -1, 0, 1, so even if the sum turns out to be -2 or 2, `cmp()` will still return -1 or 1, respectively. A value of 0 is returned if both sets of numbers and strings are the same, or if the comparisons offset each other, i.e., `(n1 < n2)` and `(s1 > s2)` or vice versa.

Given the above criteria, we present the code below for `numstr.py:`

### Example 13.2. Emulating Types with Classes (`numstr.py`)

```
 <$nopage>
001 1  # !/usr/bin/env python
002 2
003 3  class NumStr:
004 4
005 5      def __init__(self, num=0, string=''):# constr.
006 6          self.__num = num
007 7          self.__string = string
008 8
009 9      def __str__(self):                   # define for str()
010 10         return \xd4 [%d :: %s]' % \
011 11             self.__num, \xd4 self.__string\xd4 )
012 12     __repr__ = __str__
013 13
014 14     def __add__(self, other):            # define for s+o
015 15         if isinstance(other, NumStr):
016 16             return self.__class__(self.__num + \
```

```
017 17                 other.__num, \
018 18                 self.__string + other.__string)
019 19         else: <$nopage>
020 20             raise TypeError, \
021 21     'illegal argument type for built-in operation'
022 22
023 23     def __radd__(self, other):           # define for o+s
024 24         if isinstance(other, NumStr):
025 25             return self.__class__(other.num + \
026 26                 self.num, other.str + self.str)
027 27         else: <$nopage>
028 28             raise TypeError, \
029 29     'illegal argument type for built-in operation'
030 30
031 31     def __mul__(self, num):              # define for o*n
032 32         if type(num) == type(0):
033 33             return self.__class__(self.__num * num,\
034 34                 self.__string * num)
035 35         else: <$nopage>
036 36             raise TypeError, \
037 37     'illegal argument type for built-in operation'
038 38
039 39     def __nonzero__(self):               # reveal tautology
040 40         return self.__num or len(self.__string)
041 41
042 42     def __norm_cval(self, cmpres): # normalize cmp()
043 43         return cmp(cmpres, 0)
044 44
045 45     def __cmp__(self, other):             # define for cmp()
046 46         nres = self.__norm_cval(cmp(self.__num, \
047 47             other.__num))
048 48         sres = self.__norm_cval(cmp(self.__string, \
049 49             other.__string))
050 50
051 51     if not (nres or sres): return 0 # both 0
052 52         sum  = nres + sres
053 53     if not sum: return None # one <,one>
054 54         return sum
055  <$nopage>
```

Here is an example execution of how this class works:

```
>>> a = NumStr(3, 'foo')
>>> b = NumStr(3, 'goo')
>>> c = NumStr(2, 'foo')
>>> d = NumStr()
>>> e = NumStr(string='boo')
>>> f = NumStr(1)
>>> a
[3 :: 'foo']
>>> b
[3 :: 'goo']
>>> c
[2 :: 'foo']
>>> d
```

```
[0 :: '']
>>> e
[0 :: 'boo']
>>> f
[1 :: '']
>>> a < b
1
>>> b < c
0
>>> a == a
1
>>> b * 2
[6 :: 'googoo']
>>> a * 3
[9 :: 'foofoofoo']
>>> e + b
[3 :: 'boogoo']
>>> if d: 'not false'
…
>>> if e: 'not false'
…
'not false'
>>> cmp(a,b)
-1
>>> cmp(a,c)
1
>>> cmp(a,a)
0
```

### Line-by-line Explanation

### Lines 3–7

The constructor `__init__()` function sets up our instance initializing itself with the values passed in to the class instantiator `NumStr()`. If either value is missing, the attribute takes on the default false value of either zero or the empty string, depending on the argument.

One significant oddity is the use of double underscores to name our attributes. As we will find out in the next section, this is used to enforce a level, albeit elementary, of privacy. Programmers importing our module will not have straightforward access to our data elements. We are attempting to enforce one of the encapsulation properties of OO design by permitting access only though accessor functionality. If this syntax appears odd or uncomfortable to you, you can remove all double underscores from the instance attributes, and the examples will still work exactly in the same manner.

All attributes which begin with a double underscore ( __ ) are "mangled" so that these names are not as easily accessible during run-time. They are not, however, mangled in such a way so that it cannot be easily reverse-engineered. In fact, the mangling pattern is fairly well-known and easy to spot. The main point is to prevent the name from being

accidentally used when being imported by an external module where conflicts may arise. The name is changed to a new identifier name containing the class name to ensure that it does not get "stepped on" unintentionally. For more information, check out Section 13.14 on privacy.

### Lines 9–12

We choose the string representation of our ordered pair to be "`[num :: 'str']`" so it is up to `__str__()` to provide that representation whenever `str()` is applied to our instance and when the instance appears in a **`print`** statement. Because we want to emphasize that the second element is a string, it is more visually convincing if the users view the string surrounded by quotation marks. To that end, we call `repr()` using the single back quotation marks to give the evaluatable version of a string, which does have the quotation marks:

```
>>> print a
[3 :: 'foo']
```

*Not* calling `repr()` on `self.__string` (leaving the back quotations off) would result in the string quotations being absent. For the sake of argument, let us effect this change for learning purposes. Removing the backquotes, we edit the **`return`** statement so that it now looks like this:

```
return '[%d :: %s]' % (self.__num, self.__string)
```

Now calling **`print`** again on an instance results in:

```
>>> print a
[3 :: foo]
```

How does that look without the quotations? Not as convincing that "foo" is a string, is it? It looks more like a variable. The author is not as convinced either. (We quickly and quietly back out of that change and pretend we never even touched it.)

The first line of code after the `__str__()` function is the assignment of that function to another special method name, `__repr__`. We made a decision that an evaluatable string representation of our instance should be the same as the printable string representation. Rather than defining an entirely new function which is a duplicate of `__str__()`, we just create an alias, copying the reference.

When you implement `__str__()`, it is the code that is called by the interpreter if you ever apply the `str()` built-in function using that object as an argument. The same goes for `__repr__()` and `repr()`.

How would our execution differ if we chose not to implement `__repr__()`? If the assignment is removed, only the **`print`** statement (which calls `str()` will show us the contents of our object. The evaluatable string representation defaults to the Python standard of `<…some_object_information…>`.

```
>>> print a    # calls str(a)
[3 :: 'foo']
>>> a          # calls repr(a)
<NumStr.NumStr instance at 122640>
```

### Lines 14–29

One feature we would like to add to our class is the addition operation, which we described earlier. One of Python's features as far as customizing classes goes is the fact that we can overload operators to make these types of customizations more "realistic." Invoking a function such as "`add(obj1, obj2)`" to "add" objects `obj1` and `obj2` may *seem* like addition, but is it not more compelling to be able to invoke that same operation using the plus sign ( `+` ) like this? `? obj1 + obj2`

Overloading the plus sign requires the implementation of two functions, `__add__()` and `__radd__()`, as explained in more detail in the previous section. The `__add__()` function takes care of the SELF + OTHER case, but we need to define `__radd__()` to handle the OTHER + SELF scenario. The numeric addition is not affected as much as the string concatenation is because order matters.

The addition operation adds each of the two components, with the pair of results forming a new object—created as the results are passed to a call for instantiation as calling `self.__class__()` (again, also previously explained above). Any object other than a like type should result in a `TypeError` exception, which we raise in such cases.

### Lines 31–37

We also overload the asterisk [by implementing `__mul__()`] so that both numeric multiplication and string repetition are performed, resulting in a new object, again created via instantiation. Since repetition allows only an integer to the right of the operator, we must enforce this restriction as well. We also do not define `__rmul__()` for the same reason.

### Lines 39–40

Python objects have a concept of having a Boolean value at any time. For the standard types, objects have a false value when they are either a numeric equivalent of zero or an empty sequence or mapping. For our class, we have chosen that both its numeric value must be zero *and* the string empty in order for any such instance to have a false value. We override the `__nonzero__()` method for this purpose. Other objects such as those which strictly emulate sequence or mapping types use a length of zero as a false value. In those cases, you would implement the `__len__()` method to effect that functionality.

**Lines 39–54**

`__norm_cval()` is not a special method. Rather, it is a helper function to our overriding of `__cmp__();` its sole purpose is to convert all positive return values of `cmp()` to 1, and all negative values to -1. `cmp()` normally returns arbitrary positive or negative values (or zero) based on the result of the comparison, but for our purposes, we need to restrict the return values to only -1, 0, and 1. Calling `cmp()` with integers will give us the result we need, being equivalent to the following snippet of code:

```
def __norm_cval(self, cmpres):
    if cmpres < 0:
        return -1
    elif cmpres > 0:
        return 1
    else:
        return 0
```

The actual comparison of two like objects consists of comparing the numbers and the strings, and returning the sum of the comparisons. You may have noticed in the code above that we prepended a double underscore ( __ ) in front of our data attributes. This directive provides a light form of privacy.

## Privacy

Attributes in Python are, by default, "public" all the time, accessible by both code within the module and modules that import the module containing the class.

Many OO languages provide some level of privacy for the data and provide only accessor functions to provide access to the values. This is known as implementation hiding and is a key component to the encapsulation of the object. Most OO languages provide "access specifiers" to restrict who has access to member functions.

Python 1.5 introduces an elementary form of privacy for class elements (attributes or methods). Attributes which begin with a double underscore (__) are mangled during run-time so direct access is thwarted. In actuality, the name is prepended with an underscore followed by the class name. For example, let us take the `self.__num` attribute found in Example 13.2 (`numstr.py`). After the mangling process, the identifier used to access that

data value is now `self._NumStr__num`. Adding the class name to the newly-mangled result will prevent it from clashing with the same name in either ancestor or descendant classes.

Although this provides some level of privacy, the algorithm is also in the public domain and can be defeated easily. It is more of a protective mechanism for importing modules that do not have direct access to the source code or for other code within the same module.

One way to prevent the source code from being accessed is to allow only access to the byte-compiled `.pyc` file. For example, a software company shipping Python software may choose to provide only the `.pyc` files. This helps to ensure that no one can maliciously gain programmatic access to private variables and methods.

As we discovered in Chapter 12, simple module-level privacy is provided by using a single underscore character prefixing an attribute name. This prevents a module attribute from being imported with "`from mymodule import *`".

## Delegation

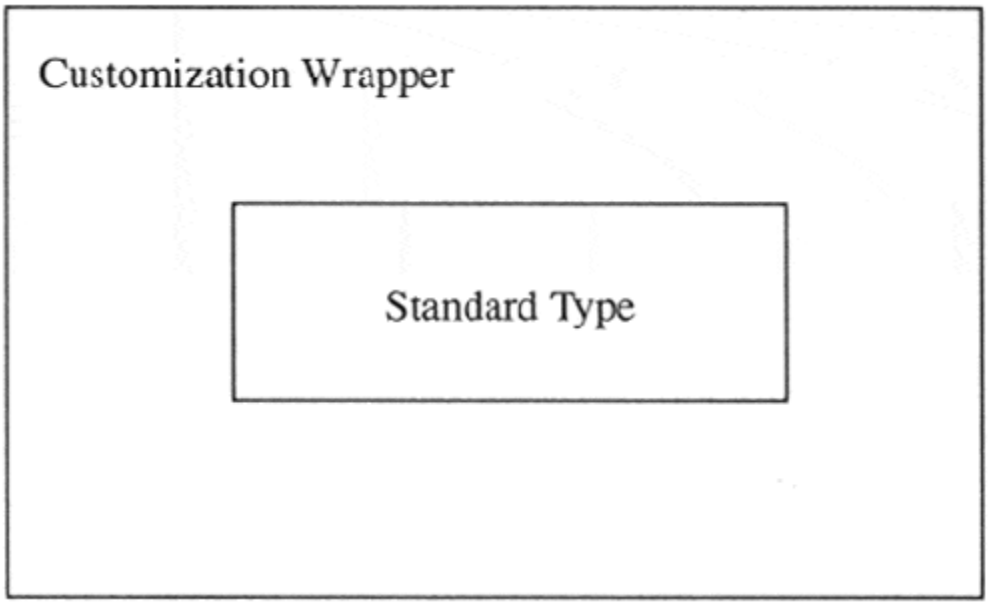
### Wrapping

"Wrapping" is a term you will hear often in the Python programming world. It is a generic moniker to describe the packaging of an existing object, whether it be a data type or a piece of code, adding new, removing undesired, or otherwise modifying existing functionality to the existing object.

The subclassing or derivation of a standard type in Python is not allowed; however, you can wrap any type as the core member of a class so that the new object's behavior mimics all existing behavior of the data type that you want, does not do what you do not want it to do, and perhaps does something a little extra. This is called "wrapping a type." In the Appendix, we will discuss how to extend Python, another form of wrapping.

Wrapping consists of defining a class whose instances have the core behavior of a standard type. In other words, it not only sings and dances now, but also walks and talks like our original type. Figure13-2 attempts to illustrate what a type wrapped in a class looks like. The core behavior of a standard type is in the center of the figure, but it is also enhanced by new or updated functionality, and perhaps even by different methods of accessing the actual data.

**Figure 13.2. Wrapping a Type**




**Class Object (which behaves like a type)**

You may also wrap classes, but this does not make as much sense because there is already a mechanism for taking an object and wrapping it in a manner as described above for a standard type. How would you take an existing class, mimic the behavior you desire, remove what you do not like, and perhaps tweak something to make the class perform differently from the original class? That process, as we discussed recently, is derivation. We wrap only types because they cannot be subclassed.

## Implementing Delegation

Delegation is a characteristic of wrapping that you can utilize which simplifies the process with regards to dictating functionality.

Delegation is a form of wrapping which takes advantage of pre-existing functionality to maximize code reuse. Wrapping a type generally consists of some sort of customization to the existing type. As we mentioned before, this "tweaking" comes in the form of new, modified, or removed functionality compared to the original product. Everything else should "remain the same," or keep its existing functionality and behavior. Delegation is the process whereby all the updated functionality is handled as part of the new class, but the existing functionality is "delegated" to the default attributes of the object.

The key to implementing delegation is to override the `__getattr__()` method with code containing a call to the built-in `getattr()` function. Specifically, `getattr()` is invoked to obtain the default object attribute (data attribute or method) and return it for access or invocation. The way the special method `__getattr__()` works is that when an attribute is searched for, any local ones are found first (the customized ones). If the search fails, then `__getattr__()` is invoked, which then calls `getattr()` to obtain an object's default behavior.

In other words, when an attribute is referenced, the Python interpreter will attempt to find that name in the local namespace, such as a customized method or local instance attribute. If it is not found in the local dictionary, then the class namespace is searched, just in case a class attribute was accessed. Finally, if both searches fail, the hunt begins to delegate the request to the original object, and that is when `__getattr__()` is invoked.

### Simple Example Wrapping Any Object

Let us take a look at an example. We present below a class which wraps nearly any object, providing basic functionality as string representations with `repr()` and `str()`. Additional customization comes in the form of the `get()` method, which removes the wrapping and returns the raw object. All remaining functionality is delegated to the object's native attributes as retrieved by `__getattr__()` when necessary.

Here's an example wrapping class:

```
class WrapMe:

     def __init__(self, obj):
         self.__data = obj

     def get(self):
         return self.__data

     def __repr__(self):
         return 'self.__data'

     def __str__(self):
         return str(self.__data)

     def __getattr__(self, attr):
         return getattr(self.__data, attr)
```

In our first example, we will use complex numbers, because of all Python's numeric types, complex numbers are the only one with attributes, data attributes as well as its `conjugate()` built-in method. Remember that attributes can be both data attributes as well as functions or methods. Again, we chose complex numbers because it is an example of a standard which has both attribute types. Here is an example with a complex number:

```
>>> wrappedComplex = WrapMe(3.5+4.2j)
>>> wrappedComplex             # wrapped object [repr()]
[repr()]
(3.5+4.2j)
>>> wrappedComplex.real        # real attribute
3.5
>>> wrappedComplex.imag        # imaginary attribute
42.2
>>> wrappedComplex.conjugate() # conjugate() method
```

```
(3.5-4.2j)
>>> wrappedComplex.get()        # actual object
(3.5+4.2j)
```

Once we create our wrapped object type, we obtain a string representation, silently using the call to `repr()` by the interactive interpreter. We then proceed to access all three complex number attributes, none of which is defined for our class. All three accesses are delegated to the object's attributes via the `getattr()` method. The final access to our example object is to retrieve an attribute that is defined for our object, the `get()` method which returns the actual data object that we wrapped.

Our next example using our wrapping class uses a list. We will create the object, then perform multiple operations, delegating each time to list methods.

```
>>> wrappedList = WrapMe([123, 'foo', 45.67])
>>> wrappedList.append(\xd4 bar\xd5 )
>>> wrappedList.append(123)
>>> wrappedList
[123, 'foo', 45.67, 'bar', 123]
>>> wrappedList.index(45.67)
2
>>> wrappedList.count(123)
2
>>> wrappedList.pop()
123
>>> wrappedList
[123, 'foo', 45.67, 'bar']
```

Notice that although we are using a class instance for our examples, they exhibit behavior extremely similar to the data types which they wrap. Be aware, however, that only existing attributes can delegated.

Special behaviors which are not in a type's method list will not be accessible since they are not attributes. One example is the slicing operations of lists which are built-in to the type and not available as an attribute like the `append()` method for example. Another way of putting it is that the slice operator ( `[ ]` ) is part of the sequence type and is not implemented through the `__getitem__()` special method.

```
>>> wrappedList[3]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "wrapme.py", line 21, in __getattr__
    return getattr(self.data, attr)
AttributeError: __getitem__
```

The `AttributeError` exception results from the fact that the slice operator invokes the `__getitem__()` method, and `__getitem__()` is not defined as a class instance method nor is it a method of list objects. Recall that `getattr()` is called only when an exhaustive search through an instance's or class's dictionaries fails to find a successful match. As you can see above, the call to `getattr()` is the one which fails, triggering the exception.

However, we can always cheat by accessing the real object [with our `get()` method] and its slicing ability instead:

```
>>> realList = wrappedList.get()
>>> realList[3]
'bar'
```

You probably have a good idea now why we implemented the `get()` method—just for cases like this where we need to obtain access to the original object. We can bypass assigning local variable (`realList`) by accessing the attribute of the object directly from the access call:

```
>>> wrappedList.get()[3]
'bar'
```

The `get()` method returns the object which is then immediately indexed to obtain the sliced subset.

```
>>> f = WrapMe(open('/etc/motd'))
>>> f
<open file '/etc/motd', mode 'r' at 80e95e0>
>>> f.readline()
'Have a lot of fun…\012'
>>> f.tell()
21
>>> f.seek(0)
>>> print f.readline(),
Have a lot of fun…
>>> f.close()
>>> f
<closed file '/etc/motd', mode 'r' at 80e95e0>
```

Once you become familiar with an object's attributes, you begin to understand where certain pieces of information originate and are able to duplicate functionality with your newfound knowledge:

```
>>> print "<%s file %s, mode %s at %x>"  % \
… (f.closed and 'closed' or 'open', "f.name",
"f.mode", id(f.get()))
<closed file '/etc/motd', mode 'r' at 80e95e0>
```

This concludes the sampling of our simple wrapping class. We have only just begun to touch on class customization with type emulation. You will discover that there are an infinite number of enhancements you can make to further increase the usefulness of your code. One such enhancement would be to add timestamps to objects. In the next subsection, we will add another dimension to our wrapping class: time.

### Updating Our Simple Wrapping Class

Creation time, modification time, and access time are familiar attributes of files, but nothing says that you cannot add this type of information to objects. After all, certain applications may benefit from these additional pieces of information.

If you are unfamiliar with using these three pieces of chronological data, we will attempt to clarify them. The creation time (or "ctime") is the time of instantiation, the modification time (or "mtime") refers to the time that the core data was updated [accomplished by calling the new `set()` method], and the access time (or "atime") is the timestamp of when the data value of the object was last retrieved or an attribute was accessed.

Proceeding to updating the class we defined earlier, we create the module `twrapme.py`, given in Example 13.3.

How did we update the code? Well, first, you will notice the addition of three new methods: `gettimeval()`, `gettimestr()`, and `set()`. We also added lines of code throughout which update the appropriate timestamps based on the type of access performed.

The `gettimeval()` method takes a single character argument, either "c," "m," or "a," for create, modify, or access time, respectively, and returns the corresponding time that is stored as a float value. `gettimestr()` simply returns a pretty-printable string version of the time as formatted by the `time.ctime()` function.

Let us take a test drive of our new module. We have already seen how delegation works, so we are going to wrap objects without attributes to highlight the new functionality we just added.

### Example 13.3. Wrapping Standard Types (`twrapme.py`)

Class definition which wraps any built-in type, adding time attributes; `get()`, `set()`, and string representation methods; and delegating all remaining attribute access to those of the standard type.

<$nopage>

```
001 1  #!/usr/bin/env python
002 2
003 3  from time import time, ctime
004 4
005 5  class TimedWrapMe:
006 6
007 7      def __init__(self, obj):
008 8         self.__data = obj
009 9         self.__ctime = self.__mtime = \
010 10        self.__atime = time()
011 11
012 12     def get(self):
013 13        self.__atime = time()
014 14        return self.__data
015 15
016 16     def gettimeval(self, t_type):
017 17        if type(t_type) != type('') or \
018 18                t_type[0] not in 'cma':
019 19             raise TypeError, \
020 20             "argument of 'c', 'm', or 'a' req'd"
021 21        return eval('self._%s__%stime' % \
022 22             (self.__class__.__name__, t_type[0]))
023 23
024 24     def gettimestr(self, t_type):
025 25        return ctime(self.gettimeval(t_type))
026 26
027 27     def set(self, obj):
028 28        self.__data = obj
029 29        self.__mtime = self.__atime = time()
030 30
031 31     def __repr__(self):# rep()
032 32        self.__atime = time()
033 33        return \xd4 self.__data\xd4
034 34
035 35     def __str__(self):# str()
036 36        self.__atime = time()
037 37        return str(self.__data)
038 38
039 39     def __getattr__(self, attr):# delegate
040 40        self.__atime = time()
041 41        return getattr(self.__data, attr)
042  <$nopage>
```

```
>>> timeWrappedObj = TimedWrapMe(932)
>>> timeWrappedObj.gettimestr('c')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('m')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('a')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj
932
>>> timeWrappedObj.gettimestr('c')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('m')
```

```
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('a')
'Wed Apr 26 20:48:05 2000'
```

You will notice that when an object is first wrapped, the creation, modification, and last access times are all the same. Once we access the object, the access time is updated, but not the others. If we use `set()` to replace the object, the modification and last access times are updated. One final read access to our object concludes our example.

```
>>> timeWrappedObj.set('time is up!')
>>> timeWrappedObj.gettimestr('m')
'Wed Apr 26 20:48:35 2000'
>>> timeWrappedObj
'time is up!'
>>> timeWrappedObj.gettimestr('c')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('m')
'Wed Apr 26 20:48:35 2000'
>>> timeWrappedObj.gettimestr('a')
'Wed Apr 26 20:48:46 2000'
```

### Wrapping a Specific Object with Enhancements

The next example represents a class which wraps a file object. Our class will behave exactly in the same manner as a regular file object with one exception: In write mode, only strings in all capital letters are written to the file.

The problem we are trying to solve here is for a case where you are writing text files whose data is to be read by an old mainframe computer. Many older style machines are restricted to uppercase letters for processing, so we want to implement a file object where all text written to the file is automatically converted to uppercase without the programmer's having to worry about it. In fact, the only noticeable difference is that rather than using the `open()` built-in function, a call is made to instantiate the `capOpen` class. Even the parameters are exactly the same as for `open()`.

Example 13.4 represents that code, written as `capOpen.py`. Let us take a look at an example of how to use this class:

### Example 13.4. Wrapping a File Object (`capOpen.py`)

This class extends on the example from Python FAQ 4.48, providing a file-like object which customizes the `write()` method while delegating the rest of the functionality to the file object.

```
                                                                    <$nopage>
001 1  #!/usr/bin/env python
```

```
002 2
003 3  from string import upper
004 4
005 5  class capOpen:
006 6      def __init__(self, fn, mode='r', buf=-1):
007 7          self.file = open(fn, mode, buf)
008 8
009 9      def __str__(self):
010 10         return str(self.file)
011 11
012 12     def __repr__(self):
013 13         return 'self.file'
014 14
015 15     def write(self, line):
016 16         self.file.write(upper(line))
017 17
018 18     def __getattr__(self, attr):
019 19         return getattr(self.file, attr)
020  <$nopage>
```

```
>>> f = capOpen('/tmp/xxx', 'w')
>>> f.write('delegation example\n')
>>> f.write('faye is good\n')
>>> f.write('at delegating\n')
>>> f.close()
>>> f
<closed file '/tmp/xxx', mode 'w' at 12c230>
```

As you can see above, the only call out of the ordinary is the first one to `capOpen()` rather than `open()`. All other code is identical to what you would do if you were interacting with a real file object rather than a class instance which behaves like a file object. All attributes other than `write()` have been delegated to the file object. To confirm the success of our code, we load up the file and display its contents. (Note that we can use either `open()` or `capOpen()`, but chose only `capOpen()` because we have been working with this example.)

```
>>> f = capOpen('/tmp/xxx')
>>> allLines = f.readlines()
>>> for eachLine in allLines:
…       print eachLine,
…
DELEGATION EXAMPLE
FAYE IS GOOD
AT DELEGATING
```

## Related Modules and Documentation

Python has several classes which extend the existing functionality of the core language which we have described in this chapter. The `User*` modules are like pre-cooked meals,

ready to eat. We mentioned how classes have special methods which, if implemented, can customize classes so that when wrapped around a standard type, they can give instances type-like qualities.

`UserList` and `UserDict`, along with the new `UserString` (introduced in Python 1.6), represent modules that define classes that act as wrappers around list, dictionary, and string objects, respectively. The primary objective of these modules is to provide the desired functionality for you so that you do not have to implement them yourself, and to serve as base classes which are appropriate for subclassing and further customization. Python already provides an abundance of useful built-in types, but the added ability to perform "built-it yourself" typing makes it an even more powerful language.

In Chapter 4, we introduced Python's standard as well as other built-in types. The `types` module is a great place to learn more about Python's types as well as those which are out of the scope of this text. The `types` module also defines type objects which can be used to make comparisons. (Such comparisons are popular in Python because it does not support method overloading—this keeps the language simple, yet there are tools that add functionality to a part of the language where it had appeared to be lacking.)

The following piece of code checks to see if the object `data` is passed into the `foo` function as an integer or string, and does not allow any other type (raises an exception):

```
def foo(data):
    if type(data) == type(0):
        print 'you entered an integer'
    elif type(data) == type(''):
        print 'you entered a string'
    else:
        raise TypeError, 'only integers or strings!'
```

Although the above code is effective, you may also use attributes of the `types` module instead for more clarity:

```
from types import *
def foo(data):
    if type(data) == IntType:
        print 'you entered an integer'
    elif type(data) == StringType:
        print 'you entered a string'
    else:
        raise TypeError, 'only integers or strings!'
```

The last related module is the `operator` module. This module provides functional versions of most of Python's standard operators. There may be occasions where this type of interface proves more versatile than hard-coding use of the standard operators.

Given below is one example. As you look through the code, imagine the extra lines of code which would have been required if individual operators had been part of the implementation:

```
>>> from operator import *          # import all operators
>>> vec1 = [12, 24]
>>> vec2 = [2, 3, 4]
>>> opvec = (add, sub, mul, div)    # using +, -, *, /
>>> for eachOp in opvec:            # loop thru operators
…         for i in vec1:
…             for j in vec2:
…                print '%s(%d, %d) = %d' % \
…                      (eachOp.__name__, i, j, eachOp(i, j))
…
add(12, 2) = 14
add(12, 3) = 15
add(12, 4) = 16
add(24, 2) = 26
add(24, 3) = 27
add(24, 4) = 28
sub(12, 2) = 10
sub(12, 3) = 9
sub(12, 4) = 8
sub(24, 2) = 22
sub(24, 3) = 21
sub(24, 4) = 20
mul(12, 2) = 24
mul(12, 3) = 36
mul(12, 4) = 48
mul(24, 2) = 48
mul(24, 3) = 72
mul(24, 4) = 96
div(12, 2) = 6
div(12, 3) = 4
div(12, 4) = 3
div(24, 2) = 12
div(24, 3) = 8
div(24, 4) = 6
```

The code snippet above defines three vectors, two containing operands and the last representing the set of operations the programmer wants to perform on each pair of available operands. The outermost loop iterates through each operation while the inner pair of loops creates every possible combination of ordered pairs from elements of each operand vector. Finally, the print statement simply applies the current operator with the given arguments.

A list of the modules we described above is given in Table 13.5.

**Table 13.5. Class Related Modules**

| Module | Description |
|---|---|
| `UserList` | provides a class wrapper around list objects |
| `UserDict` | provides a class wrapper around dictionary objects |
| `UserString`[a] | provides a class wrapper around string objects; also included is a `MutableString` subclass which provides that kind of functionality, if so desired |
| `types` | defines names for all Python object types as used by the standard Python interpreter |
| `operator` | processes site-specific modules or packages |

[a] new in Python 1.6

There are plenty of class and object-oriented programming related questions in the Python FAQ. In Section 13.5.3, we noted how it was dangerous to track instances by keeping references to them. For more information on this phenomenon, see Python FAQ 4.17. Most of the relevant questions on classes and object-oriented programming are found in Sections 4 and 6 of the FAQ.

The code of Example 13.4 is inspired by the code implemented in Python FAQ 4.48. This FAQ includes a short example of wrapping a file object and modifying the `write()` method. Our version represents a complete class that can be used like a file object rather than simply passing in an existing (and open) file object. The one caveat to our class is that it applies only to files which use the `write()` method; therefore, we also refer the reader to Exercise 13–16, where the `writelines()` method is also implemented.

Finally, we point out again, the Python Library and Language Reference manuals are invaluable sources of related material.

## Exercises

**1:**

*Programming.* Name some benefits of object-oriented programming over older forms of programming.

**2:**

*Functions vs. Methods.* What are the differences between functions and methods?

**3:**

*Customizing Classes.* Create a class to format floating point values to monetary amounts. In this exercise, we will use United States currency, but feel free to implement your own.

Preliminary work: Create a function called `dollarize()` which takes a floating point value and returns that value as a string properly formatted with symbols and rounded to obtain a financial amount. For example: `dollarize(1234567.8901) ?` '$1,234,567.89' The `dollarize()` function should allow for commas, such as 1,000,000, and dollar signs. Any negative sign should appear to the *left* of the dollar sign. Once you have completed this task, then you are ready to convert it into a useful class called `MoneyFmt.`

The `MoneyFmt` class contains a single data value, the monetary amount, and has five methods (feel free to create your own outside of this exercise). The `__init__()` constructor method initializes the data value, the update() method replaces the data value with a new one, the `__nonzero__()` method is Boolean, returning 1 (true) if the data value is non-zero, the `__repr__()` method returns the amount as a float, and the `__str__()` method displays the value in the string-formatted manner that `dollarize()` does.

(a) Fill in the code to the `update()` method so that it will update the data value.

(b) Use the work you completed for `dollarize()` to fill in the code for the `__str__()` method.

(c) Fix the bug in the `__nonzero__()` method, which currently thinks that any value less than one, i.e., fifty cents ($0.50), has a false value.

(d) EXTRA CREDIT: Allow the user to optionally specify an argument indicating the desire for seeing less-than and greater-than pairs for negative values rather than the negative sign. The default argument should use the standard negative sign.

You will find the code skeleton for `moneyfmt.py` presented as Example 13.5. You will find a fully-documented (yet incomplete) version of `moneyfmt.py` on the CD-ROM. If we were to import the completed class within the interpreter, execution should behave similar to the following:

```
>>> import moneyfmt
>>>
>>> cash = moneyfmt.MoneyFmt(123.45)
>>> cash
```

```
123.45
>>> print cash
$123.45
>>>
>>> cash.update(100000.4567)
>>> cash
100000.4567
>>> print cash
$100,000.46
>>>
>>> cash.update(-0.3)
>>> cash
-0.3
>>> print cash
-$0.30
>>> repr(cash)
'-0.3'
>>> 'cash'
'-0.3'
>>> str(cash)
'-$0.30'
```

### Example 13.5. Money Formatter (moneyfmt.py)

String format class designed to "wrap" floating point values to appear as monetary amounts with the appropriate symbols.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  class MoneyFmt:
004 4
005 5    def __init__(self, value=0.)  # constructor
006 6
007 7        self.value = float(value)
008 8
009 9    def update(self, value=None)  # allow updates
010 10       ###
011 11       ### (a) complete this function
012 12       ###
013 13
014 14   def __repr__(self):           # display as a float
015 15       return 'self.value'
016 16
017 17   def __str__(self):            # formatted display
018 18       val = ''
019 19
020 20       ###
021 21       ### (b) complete this function… do NOT
022 22       ###    forget about negative numbers!!
023 23       ###
024 24
```

```
025 25        return val
026 26
027 27    def __nonzero__(self):        # boolean test
028 28        ###
029 29        ### (c) find and fix the bug
030 30        ###
031 31
032 32        return int(self.value)
033  <$nopage>
```

**4:**

*User Registration.* Create a user database (login, password, and last login timestamp) class (see problems 7–5 and 9–12) which manages a system requiring users to login before access to resources is allowed. This database class manages its users, loading any previously-saved user information on instantiation and providing accessor functions to add or update database information. If updated, the database will save the new information to disk as part of its deallocation (see `__del__()`).

**5:**

*Geometry.* Create a `Point` class that consists of an ordered pair `(x, y)` representing a point's location on the X and Y axes. X and Y coordinates are passed to the constructor on instantiation and default to the origin for any missing coordinate.

**6:**

*Geometry.* Create a line/line segment class which has length and slope behaviors, in addition to the main data attributes, a pair of points (see previous problem). You should override the `__repr__()` method (and `str()`, if you want) so that the string representing a line (or line segment) are a pair of tuples, `((x1, y1), (x2, y2))`. Summary:

| | |
|---|---|
| `__repr__` `length` | display points as pair of tuples return length of line segment—do not use "len" since that is supposed tobe an integer |
| `slope` | return slope of line segment (or Noneif applicable) |

**7:**

*Date class.* Provide interface to a time module where users can request dates in a few (given) date formats such as "MM/DD/YY," "MM/DD/YYYY," "DD/MM/YY," "DD/MM/YYYY," "Mon DD, YYYY," or the standard Unix date of "Day Mon DD, HH:MM:SS YYYY." Your class should maintain a single value of date and create an instance with the given time. If not given, default to the

current time at execution. Additional methods:

| | |
|---|---|
| `update()` | changes the data value to reflect time given or current time as a default |
| `display()` | takes format indicator and displays date in requested format: |
| `'MDY' ->` | `MM/DD/YY` |
| `'MDYY' ->` | `MM/DD/YYYY` |
| `'DMY' ->` | `DD/MM/YY` |
| `'DMYY' ->` | `DD/MM/YYYY` |
| `'MODYY' ->` | `Mon DD, YYYY` |

If no format is given, default to system/`ctime()` format. EXTRA CREDIT: Merge the use of this class into Exercise 6-15.

**8:**

*`Stack` class.* A stack is a data structure with last-in-first-out (LIFO) characteristics. Think of a stack of cafeteria trays. The first one in the spring-loaded device is the last one out, and the last one in is the first one out. Your class will have the expected `push()` (add an item to the stack) and `pop()` (remove an item from the stack) methods. Add an `isempty()` Boolean method that returns 1 if the stack is empty and 0 otherwise, and a `peek()` method that returns the item at the top of the stack without popping it off.

Note that if you are using a list to implement your stacks, the `pop()` method is already available as of Python 1.5.2. Create your new class so that it contains code to detect if the `pop()` method is available. If so, call the built-in one; otherwise, it should execute your implementation of `pop()`. You should probably use a list object; if you do, do not worry about implementing any list functionality (i.e., slicing). Just make sure that your `Stack` class can perform both of the operations above correctly. See Section 13.16 and Example 6.2 for motivation.

**9:**

`Queue` *class.* A queue is a data structure that has first-in-first-out (FIFO) characteristics. A queue is like a line where items are removed from the front and added to the rear. The class should support the following methods:

`enqueue()`–adds a new element to the end of a list `dequeue()`–returns the first element and removes it from the list

See the previous problem and Example 6.3 for motivation.

**10:**

*Stacks and Queues.* Write a class which defines a data structure that can behave as both a queue (FIFO) or a stack (LIFO), somewhat similar in nature to arrays in PERL. There are four methods that should be implemented:

`shift()`–returns the first element and removes it from the list, similar to the earlier `dequeue()` function.

`unshift()`–"pushes" a new element to the front or head of the list.

`push()`–adds a new element to the end of a list, similar to the enqueue() and push() methods from previous problems.

`pop()` returns the last element and removes it from the list. It works exactly the same way as `pop()` from before.

Also see Exercises 13-8 and 13-9.

**11:**

*Electronic Commerce.* You need to create the foundations of an e-commerce engine for a B2C (business-to-consumer) retailer. You need to have a class for a customer called `User`, a class for items in inventory called `Item`, and a shopping cart class called `Cart`. Items go in `Carts`, and `Users` can have multiple Carts. Also, multiple items can go into Carts, including more than one of any single item.

**12:**

*Chat Rooms.* You have been pretty disappointed at the current quality of chat room applications and vow to create your own, start-up a new Internet company, obtain venture capital funding, integrate advertisement into your chat program, quintuple revenues in a six-month period, go public, and retire. However, none of this will happen if you do not have a pretty cool chat application.

There are three classes you would need: a `Message` class containing a message string and any additional information such as broadcast or single recipient, a `User` class that contains all the information for a person entering your chat rooms. To

really wow the VCs to get your start-up capital, you add a class `Room` that represents a more sophisticated chat system where users can create separate "rooms" within the chat area and invite others to join. EXTRA CREDIT:

Develop graphical user interface (GUI) applications for the users.

**13:**

*Stock portfolio class.* For each company, your database tracks the name, ticket symbol, purchase date, purchase price, # of shares. Methods include: Add new symbol, remove symbol, and YTD or Annual Return performance for any or all symbols given a current price (and date).

**14:**

*DOS.* Write a Unix interface shell for DOS machines. You present the user a command-line where he or she can type in Unix commands, and you interpret them and output accordingly, i.e., the "ls" command calls "dir" to give a list of filenames in a directory, "more" uses the same command (paginating through a text file), "cat" calls "type," "cp" calls "copy," "mv" calls "ren," and "rm" invokes "del," etc.

**15:**

*Delegation.* In our final comments regarding the `capOpen` class of Example 13.4 where we proved that our class wrote out the data successfully, we noted that we could use either `capOpen()` or `open()` to read the file text. Why? Would anything change if we used one or the other?

**16:**

Delegation and Functional Programming.

(a) Implement a writelines() method for the `capOpen` class of Example 13.4. Your new function will take a list of lines and write them out converted to uppercase, similar to the way the regular writelines() method differs from write(). Note that once you are done, writelines() is no longer "delegated" to the file object.

(b) Add an argument to the writelines() method that determines whether a NEWLINE should be added to every line of the list. This argument should default to a value of 0 for no NEWLINEs.

# Chapter 14. Execution Environment

There are built-ins and external modules which can provide any of the functionality described above. The programmer must decide which tool to pick from the box based on the application which requires implementation. This chapter sketches a potpourri of many of the aspects of the execution environment within Python; however, we will not discuss how to start the Python interpreter or the different command-line options. Readers seeking information on invoking or starting the Python interpreter should review Chapter 2.

Our tour of Python's execution environment consists of looking at "callable" objects and following up with a lower-level peek at code objects. We will then take a look at what Python statements and built-in functions are available to support the functionality we desire. The ability to execute other programs gives our Python script even more power, as well as being a resource-saver because certainly it is illogical to reimplement all this code, not to mention the loss of time and manpower. Python provides many mechanisms to execute programs or commands external to the current script environment, and we will run through the most common options. Next, we give a brief overview of Python's restricted execution environment, and finally, the different ways of terminating execution (other than letting a program run to completion). We begin our tour of Python's execution environment by looking at "callable" objects.

## Callable Objects

A number of Python objects are what we describe as "callable," meaning any object which can be invoked with the function operator "()". The function operator is placed immediately following the name of the callable to invoke it. For example, the function "`foo`" is called with "`foo()`". You already know this. Callables may also be invoked via functional programming interfaces such as `apply()`, `filter()`, `map()`, and `reduce()`, all of which we discussed in Chapter 11. Python has four callable objects: functions, methods, classes, and some class instances. Keep in mind that any additional references or aliases of these objects are callable, too.

### Functions

The first callable object we introduced was the function. There are three types of different function objects, the first being the Python built-in functions.

#### Built-in Functions (BIFs)

BIFs are generally written as extensions in C or C++, compiled into the Python interpreter, and loaded into the system as part of the first (built-in) namespace. As

mentioned in previous chapters, these functions are found in the `__builtin__` module and are imported into the interpreter as the `__builtins__` module. In restricted execution modes, only a subset of these functions is available. (See Section 14.6 for more details on restricted execution.)

All BIFs come with the attributes given in Table 14.1.

**Table 14.1. Built-in Function Attributes**

| *BIF Attribute* | *Description* |
|---|---|
| *bif*.__doc__ | documentation string |
| *bif*.__name__ | function name as a string |
| *bif*.__self__ | set to `None` (reserved for built-in methods) |

You can verify these attributes by using the `dir()` built-in function, as indicated below using the `type()` BIF as our example:

```
>>> dir(type)
['__doc__', '__name__', '__self__']
```

Internally, built-in functions are represented as the same type as built-in methods, so invoking the `type()` built-in function on a built-in function or method outputs "builtin_function_or_method," as indicated in the following example:

```
>>> type(type)
<type 'builtin_function_or_method'>
```

### User-defined Functions (UDFs)

The second type of function is the user-defined function. These are generally defined at the top-level part of a module and hence are loaded as part of the global namespace (once the built-in namespace has been established). Functions may also be defined in other functions; however, the function at the innermost level does not have access to the containing function's local scope. As indicated in previous chapters, Python currently supports only two scopes: the global scope and a function's local scope. All the names defined in a function, including parameters, are part of the local namespace.

All UDFs come with the attributes listed in Table 14.2.

**Table 14.2. User-defined Function Attributes**

| *UDF Attribute* | *Description* |
|---|---|
| *udf*.__doc__ | documentation string (also *udf*.func_doc) |
| *udf*.__name__ | function name as a string (also *udf*.func_name) |

| | |
|---|---|
| *udf*.func_code | byte-compiled code object |
| *udf*.func_defaults | default argument tuple |
| *udf*.func_globals | global namespace dictionary; same as calling globals(x) from within function |

Internally, user-defined functions are of the type "function," as indicated in the following example by using type():

```
>>> def foo(): pass
>>> type(foo)
<type 'function'>
```

### lambda Expressions (Functions named "<lambda>")

Lambda expressions are the same as user-defined functions with some minor differences. Although they yield function objects, lambda expressions are not created with the **def** statement and instead are created using the **lambda** keyword.

Because lambda expressions do not provide the infrastructure for naming the code which are tied to them, lambda expressions must be called either through functional programming interfaces or have their reference be assigned to a variable, and then they can be invoked directly or again via functional programming. This variable is merely an alias and is *not* the function object's name.

Function objects created by **lambda** also share all the same attributes as user-defined functions, with the only exception resulting from the fact that they are not named; the __name__ or func_name attribute is given the string "<lambda>".

Using the type() built-in function, we show that lambda expressions yield the same function objects as user-defined functions:

```
>>> lambdaFunc = lambda x: x * 2
>>> lambdaFunc(100)
200
>>> type(lambdaFunc)
<type 'function'>
```

In the example above, we assign the expression to an alias. We can also invoke type() directly on a lambda expression:

```
>>> type (lambda: 1)
<type 'function'>
```

Let's take a quick look at UDF names, using `lambdaFunc` above and `foo` from the preceding subsection:

```
>>> foo.__name__
'foo'
>>> lambdaFunc.__name__
'<lambda>'
```

## Methods

In the previous chapter, we discovered methods, functions which are defined as part of a class—these are user-defined methods. Many Python data types such as lists and dictionaries also have methods, known as built-in methods. To further show this type of "ownership," methods are named with or represented alongside the object's name via the dotted-attribute notation.

### Built-in Methods (BIMs)

We discussed in the previous section how built-in methods are similar to built-in functions. Only built-in types (BITs) have BIMs. As you can see below, the `type()` built-in function gives the same output for built-in methods as it does for built-in functions—note how we have to provide a built-in type (object or reference) in order to access a BIM:

```
>>> type([].append)
<type 'builtin_function_or_method'>
```

Furthermore, both BIMs and BIFs share the same attributes, too. The only exception is that now the `__self__` attribute points to a Python object (for BIMs) as opposed to None (for BIFs):

**Table 14.3. Built-in Method Attributes**

| *BIM Attribute* | *Description* |
|---|---|
| *bim*.__doc__ | documentation string |
| *bim*.__name__ | function name as a string |
| *bim*.__self__ | object the method is bound to |

By convention, a BIT should have the following lists of its BIMs and (built-in) attributes.

**Table 14.4. Built-in Type Attributes**

| *BIT Attribute* | *Description* |
|---|---|
| *bit*.__methods__ | list of (built-in) methods |

| *bit.__members__* | list of (built-in) data attributes |
|---|---|

Recall that for classes and instances, their data and method attributes can be obtained by using the `dir()` built-in function with that object as the argument to `dir()`. Apparently, BITs have two attributes that list their data and method attributes. Attributes of BITs may be accessed with either a reference or an actual object, as in these examples:

```
>>> aList = ['on', 'air']
>>> aList.append('velocity')
>>> aList
['on', 'air', 'velocity']
>>> aList.insert(2, 'speed')
>>> aList
['on', 'air', 'speed', 'velocity']
>>>
>>> [].__methods__
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> [3, 'headed', 'knight'].pop()
'knight'
```

It does not take too long to discover, however, that using an actual object to access its methods does not prove very useful functionally, as in the last example. No reference is saved to the object, so it is immediately garbage-collected. The only thing useful you can do with this type of access is to use it to display what methods (or members) a BIT has.

### User-defined Methods (UDMs)

User-defined methods are contained in class definitions and are merely "wrappers" around standard functions, applicable only to the class they are defined for. They may also be called by subclass instances if not overridden in the subclass definition.

As explained in the previous chapter, UDMs are associated with class objects (unbound methods), but can be invoked only with class instances (bound methods). Regardless of whether they are bound or not, all UDMs are of the same type, "instance method," as seen in the following calls to `type()`:

```
>>> class C:                                    # define class
…         def foo(self): pass                   # define UDM
…
>>> c = C()                                     # instantiation
>>> type(C.foo)                                 # type of unbound method
<type 'instance method'>
>>> type(c.foo)                                 # type of bound method
<type 'instance method'>
```

UDMs have the following attributes:

| Table 14.5. User-defined Method Attributes | |
|---|---|
| ***UDM Attribute*** | ***Description*** |
| *udm*.__doc__ | documentation string |
| *udm*.__name__ | method name as a string |
| *udm*.im_class | class which method is associated with |
| *udm*.im_func | function object for method (see UDFs) |
| *udm*.im_self | associated instance if bound, None if unbound |

Accessing the object itself will reveal whether you are referring to a bound or an unbound method. As you can also see below, a bound method reveals to which instance object a method is bound:

```
>>> C.foo                           # unbound method object
<unbound method C.foo>
>>>
>>> c.foo                           # bound method object
<method C.foo of C instance at 122c78>
>>> c                               # instance foo()'s bound to
<__main__.c instance at 122c78>
```

## Classes

The callable property of classes allows instances to be created. "Invoking" a class has the effect of creating an instance, better known as instantiation. Classes have default constructors which perform no action, basically consisting of a **pass** statement. The programmer may choose to customize the instantiation process by implementing an __init__() method. Any arguments to an instantiation call are passed on to the constructor:

```
>>> class C:
    def __init__(self, *args):
      print 'instantiated with these arguments:\n', args

>>> c1 = C()             # invoking class to instantiate c1
instantiated with these arguments:
()
>>> c2 = C('The number of the counting shall be', 3)
instantiated with these arguments:
('The number of the counting shall be', 3)
```

We are already familiar with the instantiation process and how it is accomplished, so we will keep this section brief. What is new, however, is how to make *instances* callable.

## Class Instances

Python provides the `__call__()` special method for classes which allows a programmer to create objects (instances) which are callable. By default, the `__call__()` method is not implemented, meaning that most instances are not callable. However, if this method is overridden in a class definition, instances of such a class are made callable. Calling such instance objects is equivalent to invoking the `__call__()` method. Naturally, any arguments given in the instance call are passed as arguments to `__call__()`.

You also have to keep in mind that `__call__()` is still a method, so the instance object itself is passed in as the first argument to `__call__()` as `self`. In other words, if `foo` is an instance, then `foo()` has the same effect as `foo.__call__(foo)`–the occurrence of `foo` as an argument—simply the reference to `self` that is automatically part of every method call. If `__call__()` has arguments, i.e., `__call__(self, arg)`, then `foo(arg)` is the same as invoking `foo.__call__(foo, arg)`. We present below an example of a callable instance, using a similar example to the previous section:

```
>>> class C:
…       def __call__(self, *args):
…           print "I'm callable!  Called with args:\n", args
…

>>> c = C()                                 # instantiation
>>> c                                       # our instance
<__main__.C instance at babd30>
>>> callable(c)                             # instance is callable
1
>>> c()                                     # instance invoked
I'm callable!  Called with arguments:
()
>>> c(3)                                    # invoked with 1 arg
I'm callable!  Called with arguments:
(3,)
>>> c(3, 'no more, no less')                # invoked with 2 args
I'm callable!  Called with arguments:
(3, 'no more, no less')
```

We close this subsection with a note that class instances cannot be made callable unless the `__call__()` method is implemented as part of the class definition.

## Code Objects

Callables are a crucial part of the Python execution environment, yet are only one element of a larger landscape. The grander picture consists of Python statements, assignments, expressions, even modules. These other "executable objects" do not have the ability to be invoked like callables. Rather, these objects are the smaller pieces of the puzzle which make up executable blocks of code called *code objects.*

At the heart of every callable is a code object which consists of statements, assignments, expressions, and other callables. Looking at a module means viewing one large code object which contains all the code found in the module, which can then be dissected into statements, assignments, expressions, and callables which recurse to another layer as they contain their own code objects.

In general, code objects can be executed as part of function or method invocations or using either the **exec** statement or `eval()` built-in function. A bird's eye view of a Python module also reveals a single code object representing all lines of code that make up that module.

If any Python code is to be executed, that code must first be converted to byte-compiled code (a.k.a. bytecode). This is precisely what code objects are. They do not contain any information about their execution environment, however, and that is why callables exist, to "wrap" a code object and provide that extra information.

Recall, from the previous section, the `udf.func_code` attribute for a UDFs? Well, guess what? That is a code object. Or how about the `udm.im_func` function object for UDMs? Since that is also a function object, it also has its own `udm.im_func.func_code` code object. So you can see that function objects are merely wrappers for code objects, and methods are wrappers for function objects. You can start anywhere and dig. When you get to the bottom, you will have arrived at a code object.

## Executable Object Statements and Built-in Functions

Python provides a number of built-in functions supporting callables and executable objects, including the **exec** statement. These functions let the programmer execute code objects as well as generate them using the `compile()` built-in function and are listed in Table 14.6.

**Table 14.6. Executable Object Statements and Built-in Functions**

| *Built-in Function or Statement* | *Description* |
|---|---|
| `callable(`***obj***`)` | determines if ***obj*** is callable; returns 1 if so, 0 otherwise |
| `compile(`***string, file, type***`)` | creates a code object from ***string*** of type ***type; file*** is where the code originates from (usually set to ?) |
| `eval(`***obj,*** ***globals***`=globals(),` ***locals***`=locals())` | evaluates ***obj,*** which is either a expression compiled into a code object or a string expression; global and/or local namespace dictionaries may also be provided, otherwise, the defaults for the current environment will be used |
| **exec** ***obj*** | execute ***obj,*** a single Python statement or set of statements, either in code object or string format; ***obj*** may also be a file object (opened to a valid Python script) |
| `input(`***prompt***`='')` | equivalent to `eval(raw_input(`***prompt***`=''))` |
| `intern(`***string***`)` | request intern of ***string*** |

### callable()

`callable()` is a Boolean function which determines if an object type can be invoked via the function operator ( `( )` ). It returns 1 if the object is callable and 0 otherwise. Here are some sample objects and what callable returns for each type:

```
>>> callable(dir)               # built-in function
1
>>> callable(1)                 # integer
0
>>> def foo(): pass
…
>>> callable(foo)               # user-defined function
1
>>> callable('bar')             # string
0
>>> class C: pass
…
>>> callable(C)                 # class
1
```

## compile()

`compile()` is a function which allows the programmer to generate a code object on the fly, that is, during run-time. These objects can then be executed or evaluated using the **`exec`** statement or `eval()` built-in function. It is important to bring up the point that both **`exec`** and `eval()` can take string representations of Python code to execute. When executing code given as strings, the process of byte-compiling such code must occur every time. The `compile()` function provides a one-time byte-code compilation of code so that the precompile does not have to take place with each invocation. Naturally, this is an advantage only if the same pieces of code are executed more than once. In these cases, it is definitely better to precompile the code.

All three arguments to `compile()` are required, with the first being a string representing the Python code to compile. The second string, although required, is usually set to the empty string. This parameter represents the file name (as a string) where this code object is located or can be found. Normal usage is for `compile()` to generate a code object from a dynamically-generated string of Python code—code which obviously does not read from an existing file.

The last argument is a string indicating the code object type. There are three possible values:

| | |
|---|---|
| `'eval'` | evaluatable expression [to be used with `eval()`] |
| `'single'` | single executable statement [to be used with **`exec`**] |
| `'exec'` | group of executable statements [to be used with **`exec`**] |

### Evaluatable Expression

```
>>> eval_code = compile('100 + 200', '', 'eval')
>>> eval(eval_code)
300
```

### Single Executable Statement

```
>>> single_code = compile('print "hello world!"', '', 'single')
>>> single_code
<code object ? at 120998, file "", line 0>
>>> exec single_code
hello world!
```

### Group of Executable Statements

```
>>> exec_code = compile("""
…    req = input('Count how many numbers? ')
…    for eachNum in range(req):
…        print eachNum
…    """, '', 'exec')
>>> exec exec_code
Count how many numbers? 6
0
1
2
3
4
5
```

## eval()

`eval()` evaluates an expression, either as a string representation or a pre-compiled code object created via the `compile()` built-in. This is the first argument to `eval()`. The second and third parameters, both optional, represent the objects in the global and local namespaces, respectively. If these arguments are not given, they default to objects returned by `globals()` and `locals()`, respectively. Take a look at the following example:

```
>>> eval('932')
932
>>> int('932')
932
```

We see that in this case, both `eval()` and `int()` yield the same result: an integer with the value 932. The paths they take are somewhat different, however. `eval()` takes the string in quotes and evaluates it as a Python expression. `int()` takes a string representation of an integer and converts it to an integer. It just so happens that the string consists exactly of the string 932, which as an expression yields the value 932, and that 932 is also the integer represented by the string "932." Things are not the same, however, when we use a pure string expression:

```
>>> eval('100 + 200')
300
>>> int('100 + 200')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 100 + 200
```

In this case, `eval()` takes the string and evaluates `"100 + 200"` as an expression, which, after performing integer addition, yields the value 300. The call to `int()` fails because the string argument is not a string representation of an integer—there are invalid literals in the string, namely, the spaces and "+" character.

One simple way to envision how the `eval()` function works is to imagine that the quotation marks around the expression are invisible and think, "If I were the Python interpreter, how would I view this expression?" In other words, how would the interpreter react if the same expression were entered interactively? The output after pressing the RETURN or ENTER key should be the same as what `eval()` will yield.

### exec

Like `eval()`, the `exec` statement also executes either a code object or a string representing Python code. Similarly, precompiling oft-repeated code with `compile()` helps improve performance by not having to go through the bytecode compilation process for each invocation. The `exec` statement takes exactly one argument, as indicated here with its general syntax:

**exec** *obj*

The executed object (*obj*) can be either a single statement or a group of statements, and either may be compiled into a code object (with "single" or "exec," respectively) or it can be just the raw string. Below is an example of multiple statements being sent to **exec** as a single string:

```
>>> exec """
```

```
…     x = 0
…     print 'x is currently:', x
…     while x < 5:
…        x = x + 1
…        print 'incrementing x to:', x
…     """
x is currently: 0
incrementing x to: 1
incrementing x to: 2
incrementing x to: 3
incrementing x to: 4
incrementing x to: 5
```

Finally, **exec** can also accept a valid file object to a (valid) Python file. If we take the code in the multi-line string above and create a file called `xcount.py`, then we could also execute the same code with the following:

```
>>> f = open('xcount.py')          # open the file
>>> exec f                         # execute the file
x is currently: 0
incrementing x to: 1
incrementing x to: 2
incrementing x to: 3
incrementing x to: 4
incrementing x to: 5
>>> exec f                         # try execution again
>>>                                # oops, it failed… why?
```

Note that once execution has completed, a successive call to **exec** fails. Well, it really doesn't *fail*. It just doesn't do anything, which may have caught you by surprise. In reality, **exec** has read all the data in the file and is sitting at the end-of-file (EOF). When **exec** is called again with the same file object, there is no more code to execute, so it does not do anything, hence the behavior seen above. How do we know that it is at EOF?

We use the file object's `tell()` method to tell us where we are in the file and then use `os.path.getsize()` to tell us how large our `xcount.py` script was. As you can see, there is an exact match:

```
>>> f.tell()                        # where are we in the file?
116
>>> f.close()                       # close the file
>>> from os.path import getsize
>>> getsize('xcount.py')            # what is the file size?
116
```

### Using Python to Generate and Execute Python Code Example

We now present code for the `loopmake.py` script, which is a simple computer-aided software engineering (CASE) that generates and executes loops on-the-fly. It prompts the user for the various parameters (i.e., loop type (**`while`** or **`for`**), type of data to iterate over [numbers or sequences]), generates the code string, and executes it.

### Example 14.1. Dynamically Generating and Executing Python Code (`loopmake.py`)

<$nopage>

```
001 1  #!/usr/bin/env python
002 2
003 3  dashes = '\n' + '-' * 50         # dashed line
004 4  exec_dict = {
005 5
006 6  'f': """                         # for loop
007 7  for %s in %s:
008 8      print %s
009 9  """,
010 10
011 11 's': """                         # sequence while loop
012 12 %s = 0
013 13 %s = %s
014 14 while %s < len(%s):
015 15      print %s[%s]
016 16      %s = %s + 1
017 17 """,
018 18
019 19 'n': """                         # counting while loop
020 20 %s = %d
021 21 while %s < %d:
022 22      print %s
023 23      %s = %s + %d
024 24 """
025 25 }
026 26
027 27 def main():
028 28
029 29     ltype = raw_input('Loop type? (For/While) ')
030 30     dtype = raw_input('Data type? (Number/Seq) ')
031 31
032 32     if dtype == 'n':
033 33         start = input('Starting value? ')
034 34         stop = input('Ending value (non-inclusive)? ')
035 35         step = input('Stepping value? ')
036 36         seq = str(range(start, stop, step))
037 37
038 38     else: <$nopage>
039 39         seq = raw_input('Enter sequence: ')
040 40
041 41     var = raw_input('Iterative variable name? ')
042 42
043 43     if ltype == 'f':
044 44         exec_str = exec_dict['f'] % (var, seq, var)
```

```
045 45
046 46     elif ltype == 'w':
047 47         if dtype == 's':
048 48             svar = raw_input('Enter sequence name? ')
049 59             exec_str = exec_dict['s'] % \
050 50     (var, svar, seq, var, svar, svar, var, var, var)
051 51
052 52         elif dtype == 'n':
053 53             exec_str = exec_dict['n'] % \
054 54     (var, start, var, stop, var, var, var, step)
055 55
056 56     print dashes
057 57     print 'Your custom-generated code:' + dashes
058 58     print exec_str + dashes
059 59     print 'Test execution of the code:' + dashes
060 60     exec exec_str
061 61     print dashes
062 62
063 63 if __name__ == '__main__':
064 64     main()
065  <$nopage>
```

Here are a few example executions of this script:

```
% loopmake.py
Loop type? (For/While) f
Data type? (Number/Sequence) n
Starting value? 0
Ending value (non-inclusive)? 4


Stepping value? 1
Iterative variable name? counter


-------------------------------------------------
The custom-generated code for you is:
-------------------------------------------------


for counter in [0, 1, 2, 3]:
    print counter


-------------------------------------------------
Test execution of the code:
-------------------------------------------------
0
1
2
3


-------------------------------------------------
```

```
% loopmake.py
Loop type? (For/While) w
Data type? (Number/Sequence) n
Starting value? 0
Ending value (non-inclusive)? 4
Stepping value? 1
Iterative variable name? counter


-------------------------------------------------
Your custom-generated code:
-------------------------------------------------


counter = 0
while counter < 4:
    print counter
    counter = counter + 1


-------------------------------------------------
Test execution of the code:
-------------------------------------------------
0
1
2
3


-------------------------------------------------


% loopmake.py
Loop type? (For/While) f
Data type? (Number/Sequence) s
Enter sequence: [932, 'grail', 3.0, 'arrrghhh']
Iterative variable name? eachItem


-------------------------------------------------
Your custom-generated code:
-------------------------------------------------


for eachItem in [932, 'grail', 3.0, 'arrrghhh']:
    print eachItem


-------------------------------------------------
Test execution of the code:
-------------------------------------------------
932
grail
3.0
arrrghhh
```

```
-------------------------------------------------
% loopmake.py
Loop type? (For/While) w
Data type? (Number/Sequence) s
Enter sequence: [932, 'grail', 3.0, 'arrrghhh']
Iterative variable name? eachIndex
Enter sequence name? myList


-------------------------------------------------
Your custom-generated code:
-------------------------------------------------


eachIndex = 0
myList = [932, 'grail', 3.0, 'arrrghhh']
while eachIndex < len(myList):
    print myList[eachIndex]
    eachIndex = eachIndex + 1


-------------------------------------------------
Test execution of the code:
-------------------------------------------------
932
grail
3.0
arrrghhh


-------------------------------------------------
```

### Line-by-line Explanation

#### Lines 1 – 25

In this first part of the script, we are setting up two global variables. The first is a static string consisting of a line of dashes (hence the name) and the second is a dictionary of the skeleton code we will need to use for the loops we are going to generate. The keys are "f" for a `for` loop, "s" for a `while` loop iterating through a sequence, and "n" for a counting `while` loop.

#### Lines 27 – 30

Here we prompt the user for the type of loop he or she wants and what data types to use.

#### Lines 32 – 36

Numbers have been chosen; they provide the starting, stopping, and incremental values. In this section of code, we are introduced to the `input()` built-in function for the first time. As we shall see in Section 14.3.5, `input()` is similar to `raw_input()` in that it

prompts the user for string input, but unlike `raw_input(), input()` also evaluates the input as a Python expression, rendering a Python object even if the user typed it in as a string.

**Lines 38 – 39**

A sequence was chosen; enter the sequence here as a string.

**Line 41**

Get the name of the iterative loop variable that the user wants to use.

**Lines 43 – 44**

Generate the `for` loop, filling in all the customized details.

**Lines 46 – 50**

Generate a `while` loop which iterates through a sequence.

**Lines 52– 54**

Generate a counting `while` loop.

**Lines 56 – 61**

Output the generated source code as well as the resulting output from execution of the aforementioned generated code.

**Lines 63 – 64**

Execute `main()` only if this module was invoked directly.

To keep the size of this script to a manageable size, we had to trim all the comments and error checking from the original script. You can find both the original as well as an alternate version of this script on the CD-ROM in the back of the text.

The extended version includes extra features such as not requiring enclosing quotation marks for string input, default values for input data, and detection of invalid ranges and identifiers; it also does not permit built-in names or keywords as variable names.

### `input()`

The `input()` built-in function is the same as the composite of `eval()` and `raw_input()`, equivalent to `eval(raw_input())`. Like `raw_input(), input()` has an optional parameter which represents a string prompt to display to the user. If not provided, the string has a default value of the empty string.

Functionally, `input()` differs from `raw_input()` because `raw_input()` always returns a string containing the user's input, verbatim. `input()` performs the same task of obtaining user input; however it takes things one step further by evaluating the input as a Python expression. This means that the data returned by `input()` is a Python object, the result of performing the evaluation of the input expression.

One clear example is when the user inputs a list. `raw_input()` returns the string representation of a list, while `input()` returns the actual list:

```
>>> aString = raw_input('Enter a list: ')
Enter a list: [ 123, 'xyz', 45.67 ]
>>> aString
"[ 123, 'xyz', 45.67 ]"
>>> type(aString)
<type 'string'>
```

The above was performed with `raw_input()`. As you can see, everything is a string. Now let us see what happens when we use `input()` instead:

```
>>> aList = input('Enter a list: ')
Enter a list: [ 123, 'xyz', 45.67 ]
>>> aList
[123, 'xyz', 45.67]
>>> type(aList)
<type 'list'>
```

Although the user input a string, `input()` evaluates that input as a Python object and returns the result of that expression.

## Interned Strings and `intern()`

For performance reasons, Python keeps an internal string table consisting of static string literals and identifier strings whose purpose is to speed up dictionary lookups. By keeping these strings around, any time a reference is made to either the same string literal, or to an identifier which bears a name that is in the table, no new space needs to be allocated for that string, hence saving the time it takes for memory allocation. This "interned" set of strings is not deallocated or garbage-collected until the interpreter exits.

Here is an example of an interned string:

```
>>> id('hello world')
1040072
>>> foo = 'hello world'
>>> id(foo)
```

```
1040072
>>> del foo
>>> id('hello world')
1040072
```

The string "hello world" is created in the first statement. The call to `id()` to reveal its identity was not necessary since the string was created regardless of whether or not the call was made. We did so in the example to immediately display its ID once it was created. Upon assigning this string to an identifier, we observe with another call to `id()` that, indeed, `foo` is referencing the same string object, which has been interned. If we remove the object, thereby decrementing the reference count, we see that this has no effect on the string which has been interned.

One surprising aspect may be that the string "foo" itself was interned (as the string name of an identifier). If we create two other strings, we can see that the "foo" string has an earlier ID than the newer strings, indicating that it was created first.

```
>>> id('bar')
1053088
>>> id('foo')
1052968
>>> id('goo')
1053728
```

Python 1.5 saw the debut of the `intern()` built-in function, which lets the programmer explicitly request that a string be interned. The syntax of `intern()`, as you may suspect, is:

```
intern(string)
```

The given *`string`* argument is the string to intern. `intern()` enters the string in the (global) table of interned strings. If the string is not already in the interned string table, it is interned and the string is returned. Otherwise, if the string is already there, it is simply returned.

## Executing Other (Python) Programs

When we discuss the execution of other programs, we distinguish between Python programs and all other non-Python programs, which includes binary executables or other scripting language source code. We will cover how to run other Python programs first, then how to use the `os` module to invoke external programs.

### Import

During run-time, there are a number of ways to execute another Python script. As we discussed in an earlier chapter, importing a module the first time will cause the code at the top-level of that module to execute. This is the behavior of Python importing, whether desired or not. We remind you that the only code that belongs to the top-level of a module are global variables, and class and function declarations.

These should be followed by an **if** statement that checks `__name__` to determine if a script is invoked, i.e., "**if** `__name__ == '__main__'`". In these cases, your script can then execute the main body of code, or, if this script was meant to be imported, it can run a test suite for the code in this module.

One complication arises when the imported module itself contains **import** statements. If the modules in these **import** statements have not been loaded yet, they will be loaded and their top-level code executed, resulting in recursive import behavior. We present a simple example below. We have two modules `import1` and `import2`, both with **print** statements at their outermost level. `import1` imports `import2` so that when we import `import1` from within Python, it imports and "executes" `import2` as well:

Here are the contents of `import1.py`:

```
# import1.py
print 'loaded import1'
import import2
```

And here are the contents of `import2.py`:

```
# import2.py
print 'loaded import2'
```

Here is the output when we import `import1` from Python:

```
>>> import import1
loaded import1
loaded import2
>>>
```

Following our suggested workaround of checking the value of `__name__`, we can change the code in `import1.py` and `import2.py` so that this behavior does not occur:

Here is the modified version of `import1.py`:

```
# import1.py
import import2
```

```
if __name__ == '__main__':
      print 'loaded import1'
```

The following is the code for `import2.py`, changed in the same manner:

```
# import2.py
if __name__ == '__main__'
      print 'loaded import2'
```

We no longer get any output when we import `import1` from Python:

```
>>> import import1
>>>
```

Now it does not necessarily mean that this is the behavior you should code for all situations. There may be cases where you *want* to display output to confirm a module import. It all depends on your situation. Our goal is to provide pragmatic programming examples to prevent unintended side effects.

### execfile()

It should seem apparent that importing a module is not the preferred method of executing a Python script from within another Python script; that is not what the importing process is. One side effect of importing a module is the execution of the top-level code.

Earlier in this chapter, we described how the **exec** statement can be used with a file object argument to read the contents of a Python script and execute it. This can be accomplished with the following code segment:

```
f = open(filename, 'r')
exec f
f.close()
```

The three lines can be replaced by a single call to `execfile()`:

```
execfile(filename)
```

Although the code above does execute a module, it does so only in its current execution environment (i.e., its global and local namespace). There may be a desire to execute a

module with a different set of global and local dictionaries instead of the default ones. For this purpose, we can use the `execfile()` built-in function, whose syntax allows the programmer to specify the namespaces:

```
execfile(filename, globals=globals(), locals=locals())
```

## Executing Other (Non-Python) Programs

We can also execute non-Python programs from within Python. These include binary executables, other shell scripts, etc. All that is required is a valid execution environment, i.e., permissions for file access and execution must be granted, shell scripts must be able to access their interpreter (Perl, bash, etc.), binaries must be accessible (and be of the local machine's architecture).

Finally, the programmer must bear in mind whether our Python script is required to communicate with the other program that is to be executed. Some programs require input, others return output as well as an error code upon completion (or both). Depending on the circumstances, Python provides a variety of ways to execute non-Python programs. All of the functions discussed in this section can be found in the `os` module. We provide a summary for you here in Table 14.7 (where appropriate, we annotate those which are available only for certain platforms) and introduce them to you for the remainder of this section.

As we get closer to the operating system layer of software, you will notice that the consistency of executing programs, even Python scripts, across platforms starts to get a little dicey. We mentioned above that the functions

**Table 14.7. `os` Module Functions for External Program Execution (Unix only, Windows only)**

| *os Module Function* | *Description* |
|---|---|
| `system(cmd)` | execute program `cmd` given as string, wait for program completion, and return the exit code (on Windows, the exit code is always 0) |
| `fork()` | create a child process which runs in parallel to the parent process [usually used with `exec*()`]; return twice… once for the parent and once for the child |
| `execl(file, arg0, arg1…)` | execute `file` with argument list `arg0, arg1,` etc. |
| `execv(file, arglist)` | same as `execl()` except with argument list (or tuple) `arglist` |
| `execle(file, arg0, arg1…, env)` | same as `execl()` but also providing environment variable dictionary `env` |
| `execve(file, arglist, env)` | same as `execle()` except with argument list (or tuple) `arglist` |
| `execlp(cmd, arg0, arg1…)` | same as `execl()` but search for full file pathname of cmd in |

| | user search path |
|---|---|
| `execvp(cmd, arglist)` | same as `execlp()` except with argument list (or tuple) *`arglist`* |
| `execvpe(cmd, arglist, env)` | same as `execvp()` but also providing environment variable dictionary *`env`* |
| `spawn*(mode, file, args[, env])` |  depending on *`mode`*, `spawn*()` functions can duplicate the functionality of `fork()`, `exec*()`, `system()`, `wait*()`, and/or a combination of the aforementioned |
| `popen(cmd, mode='r', buffering=-1)` |  execute *`cmd`* string, returning a file-like object as a communication handle to the running program, defaulting to read *`mode`* and default system *`buffering`* |
| `wait()` |  wait for child process to complete [usually used with `fork()` and `exec*()`] |
| `waitpid(pid, options)` |  wait for specific child process to complete [usually used with `fork()` and `exec*()`] |

described in this section are in the `os` module. Truth is, there are multiple `os` modules. For example, the one for Unix is the `posix` module. The one for Windows is `nt` (regardless of which version of Windows you are running; DOS users get the `dos` module), and the one for the Macintosh is the `mac` module. Do not worry, Python will load the correct module when you call "**`import`** `os`". You should never need to import a specific operating system module directly.

### `os.system()`

The first function on our list is `system()`, a rather simplistic function which takes a system command as a string name and executes it. Python execution is suspended while the command is being executed. When execution has completed, the exit status will be given as the return value from `system()` and Python execution resumes. This function is available for Unix and Windows only.

`system()` preserves the current standard files, including standard output, meaning that executing any program or command displaying output will be passed on to standard output. Be cautious here because certain applications such as common gateway interface (CGI) programs will cause web browser errors if output other than valid hypertext markup language (HTML) strings are sent back to the client via standard output. `system()` is generally used with commands producing no output, some of which include programs to compress or convert files, mount disks to the system, or any other command to perform a specific task that indicates success or failure via its exit status rather than communicating via input and/or output. The convention adopted is an exit status of 0 indicating success and non-zero for some sort of failure.

For the sake of providing an example, we will execute two commands which *do* have program output from the interactive interpreter so that you can observe how `system()` works.

```
>>> import os
>>> result = os.system('cat /etc/motd')
Have a lot of fun…
>>> result
0
>>> result = os.system('uname -a')
Linux solo 2.2.13 #1 Mon Nov 8 15:08:22 CET 1999 i586 unknown
>>> result
0
```

You will notice the output of both commands as well as the exit status of their execution which we saved in the `result` variable. Here is an example executing a DOS command:

```
>>> import os
>>> result = os.system('dir')


Volume in drive C has no label
Volume Serial Number is 43D1-6C8A
Directory of C:\WINDOWS\TEMP

.              <DIR>        01-08-98  8:39a .
..             <DIR>        01-08-98  8:39a ..
         0 file(s)              0 bytes
         2 dir(s)     572,588,032 bytes free
>>> result
0
```

## os.popen()

The `popen()` function is a combination of a file object and the `system()` function. It works in the same way as `system()` does, but in addition, it has the ability to establish a one-way connection to that program and then to access it like a file. If the program requires input, then you would call `popen()` with a mode of `'w'` to "write" to that command. The data that you send to the program will then be received through its standard input. Likewise, a mode of `'r'` will allow you to spawn a command, then as it writes to standard output, you can read that through your file-like handle using the familiar `read*()` methods of file object. And just like for files, you will be a good citizen and `close()` the connection when you are finished.

In one of the `system()` examples we used above, we called the Unix uname program to give us some information about the machine and operating system we are using. That command produced a line of output that went directly to the screen. If we wanted to read that string into a variable and perform internal manipulation or store that string to a log file, we could, using `popen()`. In fact, the code would look like the following:

```
>>> import os
>>> f = os.popen('uname -a')
>>> data = f.readline()
>>> f.close()
>>> print data,
Linux solo 2.2.13 #1 Mon Nov 8 15:08:22 CET 1999 i586 unknown
```

As you can see, `popen()` returns a file-like object; also notice that `readline()`, as always, preserves the newline character found at the end of a line of input text.

### os.fork(), os.exec*(), os.wait*()

Without a detailed introduction to operating systems theory, we present a "light" introduction to processes in this section. `fork()` takes your single executing flow of control known as a "process" and creates a "fork-in-the-road," if you will. The interesting thing is that your system takes *both* forks—meaning that you will have two consecutive and parallel running programs (running the same code no less because both processes resume at the next line of code immediately succeeding the `fork()` call).

The original process which called `fork()` is called the "parent" process, and the new process created as a result of the call is known as the "child process." When the child process returns, its return value is always zero; when the parent process returns, its return value is always the process identifier (a.k.a. process ID, or "PID" for short) of the child process (so the parent can keep tabs on all its children). The PIDs are the only way to tell them apart, too!

We mentioned that both processes will resume immediately after the call to `fork()`. Because the code is the same, we are looking at identical execution if no other action is taken at this time. This is usually not the intention. The main purpose for creating another process is to run another program, so we need to take divergent action as soon as parent and child return. As we stated above, the PIDs differ, so this is how we tell them apart:

The following snippet of code will look familiar to those who have experience managing processes. However, if you are new, it may be difficult to see how it works at first, but once you get it, you get it.

```
ret = os.fork()                 # spawn 2 processes, both return
if ret == 0:                    # child returns with PID of 0
    child_suite                 # child code
else:                          # parent returns with child's PID
    parent_suite                # parent code
```

The call to `fork()` is made in the first line of code. A new process called a *child process* is created. The original process is called the *parent process*. The child process has its own

copy of the virtual memory address space and contains an exact replica of the parent's address space—yes, both processes are nearly identical. Recall that `fork()` returns twice, meaning that both the parent and the child return. You might ask, how can you tell them apart if they both return? When the parent returns, it comes back with the PID of the child process. When the child returns, it has a return value of 0. This is how we can differentiate both processes.

Using an **`if-else`** statement, we can direct code for the child to execute (i.e., the **`if`** clause) as well as the parent (the **`else`** clause). The code for the child is where we can make a call to any of the `exec*()` functions to run a completely different program or some function in the same program (as long as both child and process take divergent paths of execution). The general convention is to let the children do all the dirty work while the parent either waits patiently for the child to complete its task or continues execution and checks later to see if the child finished properly.

All of the `exec*()` functions load a file or command and execute it with an argument list (either individually given or as part of an argument list). If applicable, an environment variable dictionary can be provided for the command. These variables are generally made available to programs to provide a more accurate description of the user's current execution environment. Some of the more well-known variables include the user name, search path, current shell, terminal type, localized language, machine type, operating system name, etc.

All versions of `exec*()` will replace the Python interpreter running in the current (child) process with the given file as the program to execute now. Unlike `system()`, there is no return to Python (since Python was replaced). An exception will be raised if `exec*()` fails because the program cannot execute for some reason.

The following code starts up a cute little game called "xbill" in the child process while the parent continues running the Python interpreter. Because the child process never returns, we do not have to worry about any code for the child after calling `exec*()`. Note that the command is also a required first argument of the argument list.

```
ret = os.fork()
if ret == 0:                                # child code
      execvp('xbill', ['xbill'])
else:                                       # parent code
      os.wait()
```

In this code, you also find a call to `wait()`. When children processes have completed, they need their parents to clean up after them. This task, known as "reaping a child," can be accomplished with the `wait*()` functions. Immediately following a `fork()`, a parent

can wait for the child to complete and do the clean-up then and there. A parent can also continue processing and reap the child later, also using one of the `wait*()` functions.

Regardless of which method a parent chooses, it must be performed. When a child has finished execution but has not been reaped yet, it enters a limbo state and becomes known as a "*zombie*" process. It is a good idea to minimize the number of zombie processes in your system because children in this state retain all the system resources allocated in their lifetimes, which do not get freed or released until they have been reaped by the parent.

A call to `wait()` suspends execution (i.e., waits) until a child process (any child process) has completed, terminating either normally or via a signal. `wait()` will then reap the child, releasing any resources. If the child has already completed, then `wait()` just performs the reaping procedure. `waitpid()` performs the same functionality as `wait()` with the additional arguments PID to specify the process identifier to a specific child process to wait for, and options, which is normally zero or a set of optional flags logically OR'd together. Refer to the Python, your operating system documentation, or any general operating system textbook such as Silbershatz and Galvin, Tanenbaum and Woodhull, or Stallings for more details.

### os.spawn*()

The `spawn*()` family of functions work only in the world of Windows. Depending on the mode chosen, `spawn*()` functions can duplicate the functionality of `fork()`, `exec*()`, `system()`, `wait*()`, and/or a combination of those popular Unix functions. Both `spawn()` and `spawnve()` were introduced in Python 1.5.2. For more information, go to the os module documentation in the Python Library Reference manual.

## Other Functions

Table 14.8 lists some of the functions (and their modules) which can perform some of the tasks described.

**Table 14-8. Various Functions for File Execution**

| *File Object Attribute* | W U *Description (available only on Unix or Windows platforms)* |
|---|---|
| `popen2.popen2()` | U execute a file and open file read and write access from (stdout) and to (stdin) the newly-created running program |
| `popen2.popen3()` | U execute a file and open file read and write access from (stdout and stderr) and (stdin) to the newly-created running program |
| `commands.getoutput()` | U executes a file in a subprocess, return all output as a string |

# Restricted Execution

Throughout this text, we have used only normal, unrestricted execution which provides general access to all resources available to the Python interpreter. This includes, but is not limited to: disk file or database access, establishing network connections, invoking other programs, etc.

There may be circumstances in which you want to impose restrictions on Python programs which execute on your system. Scenarios which may require you to impose restrictions on some or all of the above include: Python CGI applications, environments which allow for upload and execution of Python scripts, anonymous FTP access which has installed the Python interpreter in the `/bin` directory, etc.

There are two primary modules which aid in setting up restricted environments. The first is `Bastion`, which provides restricted access to your data. The primary method of utilizing Bastion is to instantiate the `Bastion` class around your object, providing an attribute filter with which to provide or deny access to your object's attributes.

We will not discuss `Bastion` in this text, but you may refer to the Python documentation or Beazley for more information. Instead, we will focus our efforts primarily on the `rexec` module which creates the restricted environment with which to execute untrusted Python code. We conclude our discussion of `Bastion` by stating that it can be used in conjunction with `rexec` to provide a complete and secure execution mechanism, restricting access to data as well as the run-time environment. (Both modules are installed with Python as part of the standard library.)

The `rexec` module has a primary mission to restrict the execution environment of a Python script. This module allows for a limited number of built-ins (functions and/or data attributes), imposes restrictions on which modules can imported, which attributes can and cannot be accessed from the `sys` and `os` modules, and wraps the most critical built-ins [i.e., `open()`, `reload()`, and `__import__()`] with imposed restrictions.

You will recall that the `__builtins__` module consists of all the attributes from the `__builtin__` module. If `__builtins__` *is* the `__builtin__` module, then this constitutes an unrestricted environment:

```
>>> __builtins__
<module '__builtin__' (built-in)>
```

When we impose a restricted environment, `__builtins__` will actually be a subset of the `__builtin__` module that is handpicked for the restricted environment and even becomes "inaccessible" in that environment:

```
>>> __builtins__
<module '?' (built-in)>
```

The rexec module implements an RExec class with which to subclass and create your restricted environment. This class has static members, which you override, that dictate what is and what is not allowed in the "caged" or "sandboxed" environment. We present the static data attributes of the RExec class in Table 14.9.

**Table 14-9. RExec Class Attributes**

| *Attribute Name* | *Description* |
|---|---|
| nok_builtin_names | attributes that are *not* ok to include in __builtins__ |
| ok_builtin_modules | modules that can be imported |
| ok_path | list of directories accessible in restricted environment |
| ok_posix_names | attributes that are ok to import from os module |
| ok_sys_names | attributes that are ok to import from sys module |

Instances of your RExec subclass have a number of methods with which to execute restricted code with. These are listed in Table 14.10.

**Table 14-10. RExec Class Methods**

| *Method Name* | *Description* |
|---|---|
| r_eval() | restricted version of eval() |
| r_exc_info() | restricted version of sys.exc_info() |
| r_exec() | restricted version of **exec** |
| r_execfile() | restricted version of execfile() |
| r_import() | restricted version of __import__() |
| r_open() | restricted version of open() |
| r_reload() | restricted version of reload() |
| r_unload() | restricted version of **del** module |

All of the r_*() methods except for r_exc_info() and r_open() are available as s_*(), which behaves exactly the same as their r_*() counterparts with the exception of being granted access to the standard files (standard input, output, and error). There are other methods available in rexec, and we recommend that you refer to the Python documentation for more information.

We present a small example below consisting of a pair of files. The cager.py is the program responsible for creating a restricted environment with which to safely execute another script, caged.py. The restriction we are imposing in this example is to remove all the built-in attributes and allow only a handful of them to be accessible in our restricted environment. To accomplish this, we override the nok_builtin_names attribute, a tuple listing which attributes are *not* okay in the restricted environment.

The code for cager.py is given in Example 14.2. And the code for caged.py is shown in Example 14.3.

Upon execution, we will see the output of caged.py, giving a list of built-in attributes that it *does* have access to (compare this list with the code in cager.py), as well as an

erroneous example of what happens if we call a function that we do *not* have access to, such as `eval()`:

**Example 14.2. Creating a Restricted Environment (`cager.py`)**

```
                                                    <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import rexec
004 4
005 5  class YourSandbox(rexec.RExec):
006 6      nok_builtin_names = dir(__builtins__)
007 7      nok_builtin_names.remove('dir')
008 8      nok_builtin_names.remove('str')
009 9      nok_builtin_names.remove('vars')
010 10
011 11 r = YourSandbox()
012 12 r.r_execfile("caged.py")
013  <$nopage>
```

**Example 14.3. Executing Within a Restricted Environment (`caged.py`)**

```
                                                    <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  print 'Restricted to these built-in attributes:'
004 4  for eachBI in dir(__builtin__):
005 5      print '\t', each BI:
006 6  print '\nAll others inaccessible, i.e. eval():\n'
007 7  eval (123)
008  <$nopage>


% cager.py
Restricted to these built-in attributes:
        __builtins__
        __import__
        dir
        open
        reload
        str
        vars
All others inaccessible, i.e. eval():


Traceback (most recent call last):
  File "cager.py", line 12, in ?
    r.r_execfile("caged.py")
  File "/usr/lib/python2.0/rexec.py", line 261, in r_execfile
    return execfile(file, m.__dict__)
  File "caged.py", line 7, in ?
    eval(123)
NameError: There is no variable named 'eval'
```

`r_*()` functions such as `r_open()` (and their `s_*()` equivalents) automatically direct all calls to to special wrappers which execute these functions with additional restrictions. For example, a call to `open()` calls `r_open()` which allows only read mode. Attempting to `open()` a file for write would result in an `IOError` exception:

```
IOError: can't open files for writing in restricted mode
```

You may even disallow this by overriding `r_open()`. Let us add the following method definition to our `YourSandbox` class definition in `cager.py`:

```
def r_open(f, m='r', b=-1):
          raise IOError, 'sorry, no file access period'
```

When we try to open a file for reading or writing this time, we get:

```
IOError: sorry, no file access period
```

## Terminating Execution

Clean execution occurs when a program runs to completion, where all statements in the top-level of your module finish execution and your program exits. There may be cases where you may want to exit from Python sooner, such as a fatal error of some sort. Another case is when conditions are not sufficient to continue execution.

In Python, there are varying ways to respond to errors. One is via exceptions and exception handling. Another way is to construct a "cleaner" approach so that the main portions of code are cordoned off with **`if`** statements to execute only in non-error situations, thus letting error scenarios terminate "normally." However, you may also desire to exit to the calling program with an error code to indicate that such an event has occurred.

### `sys.exit()` and `SystemExit`

The primary way to exit a program immediately and return to the calling program is the `exit()` function found in the `sys` module. The syntax for `sys.exit()` is:

```
sys.exit(status=0)
```

When `sys.exit()` is called, a `SystemExit` exception is raised. Unless monitored (in a **`try`** statement with an appropriate **`except`** clause), this exception is generally not caught nor handled, and the interpreter exits with the given status argument, which defaults to zero if not provided. `SystemExit` is the only exception which is not viewed as an error. It simply indicates the desire to exit Python.

One popular place to use `sys.exit()` is after an error is discovered in the way a command was invoked. In particular, if the arguments are incorrect, invalid, or if there are an incorrect number of them. The following Example 14.4 (`args.py`) is just a test script we created to require that a certain number of arguments be given to the program before it can execute properly:

Executing this script we get the following output:

```
% args.py
At least 2 arguments required (incl. cmd name).
usage:  args.py arg1 arg2 [arg3… ]

% args.py XXX
At least 2 arguments required (incl. cmd name).
usage:  args.py arg1 arg2 [arg3… ]

% args.py 123 abc
number of args entered: 3
args (incl. cmd name) were: ['args.py', '123', 'abc' ]

% args.py -x -2 foo bar
number of args entered: 5
args (incl. cmd name) were: ['args.py', '-x', '-2', 'foo', 'bar' ]
```

### Example 14.4. Exiting Immediately (`args.py`)

Calling `sys.exit()` causes the Python interpreter to quit. Any integer argument to `exit()` will be the returned to the caller as the exit status, which has a default value of 0.

<$nopage>

```
001 1  #!/usr/bin/env python
002 2
003 3  import sys
004 4
005 5  def usage():
006 6      print 'At least 2 arguments (incl. cmd name).'
007 7      print usage: args.py arg1 arg2 [arg3… ]'
008 8      sys.exit(1)
009 9
010 10 argc = len(sys.argv)
011 11 if argc < 3:
012 12     usage()
013 13 print "number of args entered:", argc
014 14 print "args (incl. cmd name) were:", sys.argv
```

```
015  <$nopage>
```

Many command-line driven programs test the validity of the input before proceeding with the core functionality of the script. If the validation fails at any point, a call is made to a `usage()` function to inform the user what problem caused the error as well as a usage "hint" to aid the user so that he or she will invoke the script properly the next time.

### sys.exitfunc()

`sys.exitfunc()` is disabled by default, but can be overridden to provide additional functionality which takes place when `sys.exit()` is called and before the interpreter exits. This function will not be passed any arguments, so you should create your function to take no arguments.

As described in Beazley, if `sys.exitfunc` has already been overridden by a previously defined exit function, it is good practice to also execute *that* code as part of your exit function. Generally, exit functions are used to perform some type of shutdown activity, such as closing a file or network connection, and it is always a good idea to complete these maintenance tasks, such as releasing previously held system resources.

Here is an example of how to set up an exit function, being sure to execute one if one has already been set:

```
                                                  <$nopage>
001  <$nopage>import sys
002  <$nopage>
003 prev_exit_func = getattr(sys, 'exitfunc', None)
004  <$nopage>
005  <$nopage>def my_exit_func(old_exit = prev_exit_func):
006   #       :
007   # perform cleanup
008   #       :
009   if old_exit != None and callable(old_exit):
010      old_exit()
011  <$nopage>
012 sys.exitfunc = my_exit_func
013  <$nopage>
```

We execute the old exit function after our cleanup has been performed. The `getattr()` call simply checks to see whether a previous `exitfunc` has been defined. If not, then `None` is assigned to `prev_exit_func;` otherwise, `prev_exit_func` becomes a new alias to the exiting function, which is then passed as a default argument to our new exit function, `my_exit_func.`

The call to `getattr()` could have been rewritten as:

```
                                                  <$nopage>
001  <$nopage>if hasattr(sys, 'exitfunc'):
002     prev_exit_func = sys.exitfunc       # getattr(sys, 'exitfunc')
```

```
003  <$nopage>else: <$nopage>
004     prev_exit_func = None
005  <$nopage>
```

### os._exit() Function

The _exit() function of the os module should not be used in general practice. (It is platform-dependent and available only on certain platforms anyway [Unix and Windows, to name a pair].) Its syntax is:

```
                                                    <$nopage>
001 os._exit(status)
002  <$nopage>
```

This function provides functionality opposite to that of sys.exit() and sys.exitfunc(), exiting Python immediately without performing *any* cleanup (Python or programmer-defined) at all. Unlike sys.exit(), the status argument is required. Exiting via sys.exit() is the preferred method of quitting the interpreter.

## Related Modules

In Table 14.11 you will find a list of modules other than os and sys which relate to the execution environment theme of this chapter.

**Table 14.11. Execution Environment Related Modules**

| *Module* | *Description* |
|---|---|
| popen2 | provides additional functionality on top of os.popen(): provides ability to communicate via standard files to the other process |
| commands | provides additional functionality on top of os.system(): saves all program output in a string which is returned (as opposed to just dumping output to the screen) |
| getopt | processes options and command-line arguments in such applications |
| site | processes site-specific modules or packages |
| findertools | provides an interface to Macintosh finder functionality, such as launching an application (or a document with its companion application) |

## Exercises

**1:**
*Callable Objects.* Name Python's callable objects.

**2:**
*exec vs. eval().* What is the difference between the **exec** statement and the

`eval()` built-in function?

**3:**

*`input` vs. `raw.input()`.* What is the difference between the built-in functions `input()` and `raw_input()`?

**4:**

*Execution Environment.* Create a Python script that runs other Python scripts.

**5:**

*`os.system()`.* Choose a familiar system command that performs a task without requiring input and either outputs to the screen or does not output at all. Use the `os.system()` call to run that program.

**6:**

*`commands.getoutput()`.* Solve the previous problem using `commands.getoutput()`.

**7:**

*`popen()`.Family.* Choose another familiar system command that takes text from standard input and manipulates or otherwise outputs the data. Use `os.popen()` to communicate with this program. Where does the output go? Try using `popen2.popen2()` instead.

**8:**

*Restricted Execution.* Create a restricted environment and display the contents of your `__builtins__` module to prove it.

**9:**

*Exit Function.* Design a function to be called when your program exits. Install it as `sys.exitfunc()`, run your program and show that your exit function was indeed called.

**10:**

*Shells.* Create a shell (operating system interface) program: Present a command-line interface which accepts operating system commands for execution.

EXTRA CREDIT 1: support pipes (see the dup(), dup2(), and pipe() functions in the os module). This piping procedure allows the standard output of one process to

be connected to the standard input of another.

EXTRA CREDIT 2: Support inverse pipes using parentheses, giving your shell a functional programming-like interface.

# Part II: Advanced Topics

# Chapter 15. Regular Expressions

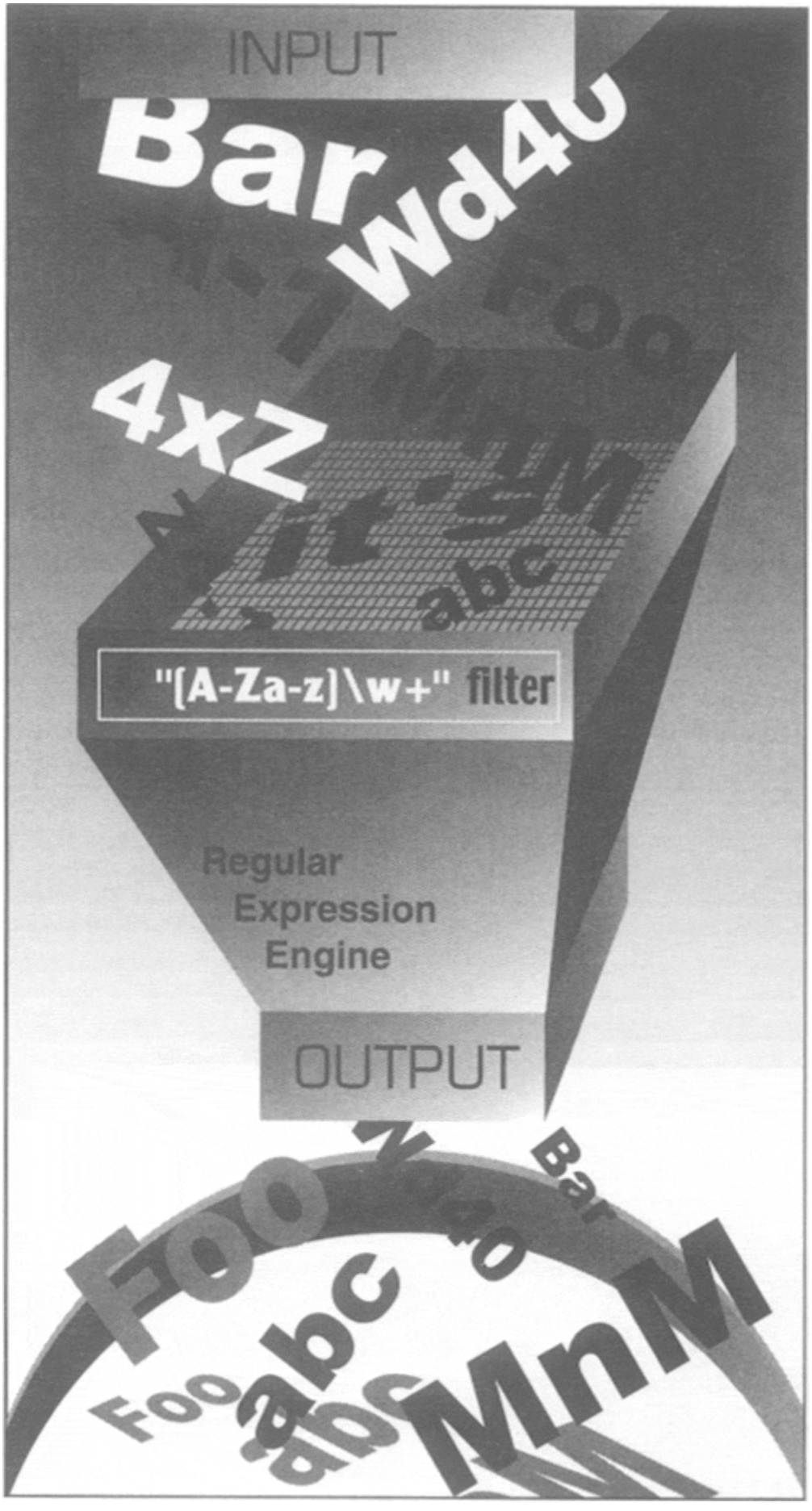
## Introduction/Motivation

Manipulating text/data is a big thing. If you don't believe me, look very carefully at what computers primarily do today. Word processing, "fill-out-form" Web pages, streams of information coming from a database dump, stock quote information, news feeds—the list goes on and on. Because we may not know the exact text or data which we have programmed our machines to process, it becomes advantageous to be able to express this text or data in patterns which a machine can recognize and take action upon.

If I were running an electronic mail (e-mail) archiving company, and you were one of my customers who requested all his or her e-mail sent and received last February, for example, it would be nice if I could set a computer program to collate and forward that information to you, rather than having a human being read through your e-mail and process your request manually. You would be horrified (and infuriated) that someone would be rummaging through your messages, even if his or her eyes were *supposed* to be looking only at timestamps. Another example request might be to look for a subject line like "ILOVEYOU" indicating a virus-infected message and remove those e-mail messages from your personal archive. So this begs the question of how can we program machines with the ability to look for patterns in text.

Regular Expressions (REs) provide such an infrastructure for advanced text pattern matching, extraction, and/or search-and-replace functionality. REs are simply strings which use special symbols and characters to indicate pattern repetition or to represent multiple characters so that they can "match" a set of strings with similar characteristics described by the pattern (Figure15.1). In other words, they enable matching of multiple strings—an RE pattern that matched only one string would be rather boring and ineffective, wouldn't you say?

**Figure 15.1. You can use regular expressions, such as the one here which recognizes valid Python identifiers. `"[A-Za-z]\w+"` means the first character should be alphabetic, i.e., either A–Z or a–z, followed by at least one (+) alphanumeric character (\w). In our filter, notice how many strings go into the filter, but the only ones to come out are the ones we asked for via the RE.**


INPUT
Bar
Wd4U
4xZ
"[A-Za-z]\w+" filter
Regular
Expression
Engine
OUTPUT
Foo
Bar
abc
MnM

Python supports REs through the standard library `re` module. In this introductory subsection, we will give you a brief and concise introduction. Due to its brevity, only the most common aspects of REs used in every day Python programming will be covered. Your experience, will of course, vary. We highly recommend reading any of the official supporting documentation as well as external texts on this interesting subject. You will never look at strings the same way again!

**NOTE**

Throughout this chapter, you will find references to searching and matching. When we are strictly discussing regular expressions with respect to patterns in strings, we will say "matching," referring to the term pattern-matching. In Python terminology, there are two main ways to accomplish pattern-matching: searching, i.e., looking for a pattern match in any part of a string, and matching, i.e., attempting to match a pattern to an entire string (starting from the beginning). Searches are accomplished using the `search()` function or method, and matching is done with the `match()` function or method. In summary, we keep the term "matching" universal when referencing patterns, and we differentiate between "searching" and "matching" in terms of how Python accomplishes pattern-matching.

## Your First Regular Expression

As we mentioned above, REs are strings containing text and special characters which describe a pattern with which to recognize multiple strings. We also briefly discussed a regular expression *alphabet* and for general text, the alphabet used for regular expressions is the set of all uppercase and lowercase letters plus numeric digits. Specialized alphabets are also possible, for instance, one consisting of only the characters "0" and "1." The set of all strings over this alphabet describes all binary strings, i.e., "0," "1," "00," "01," "10," "11," "100," etc.

Let us look at the most basic of regular expressions now to show you that although REs are sometimes considered an "advanced topic," they can also be rather simplistic. Using the standard alphabet for general text, we present some simple REs and the strings which their patterns describe. The following regular expressions are the most basic, "true vanilla," as it were. They simply consist of a string pattern which matches only one string, the string defined by the regular expression. We now present the REs followed by the strings which match them:

| RE Pattern | String(s) Matched |
|---|---|
| `foo` | `foo` |
| `Python` | `Python` |
| `abc123` | `abc123` |

The first regular expression pattern from the above chart is "foo." This pattern has no special symbols to match any other symbol other than those described, so the only string which matches this pattern is the string "foo." The same thing applies to "Python" and "abc123." The power of regular expressions comes in when special characters are used to define character sets, subgroup matching, and pattern repetition. It is these special symbols that allow a RE to match a set of strings rather than a single one.

## Special Symbols and Characters for REs

We will now introduce the most popular of the *metacharacters,* special characters and symbols, which give regular expressions their power and flexibility. You will find the most common of these symbols and characters in Table 15.1.

**Table 15.1. Common Regular Expression Symbols and Special Characters**

| *Notation* | *Description* | *Example RE* |
|---|---|---|
| ***Symbols*** | | |
| *re_string* | match literal string value *re_string* | foo |
| *re1*\|*re2* | match literal string value *re1* or *re2* | foo\|bar |
| . | match *any character* (except NEWLINE) | ::.+:: |
| ^ | match *start of string* | ^Dear |
| $ | match *end of string* | /bin/\w*sh$ |
| * | match *0 or more* occurrences of preceding RE | [A-Za-z]\w* |
| + | match *1 or more* occurrences of preceding RE | \d+\.\|\.\d+ |
| ? | match *0 or 1* occurrence(s) of preceding RE | goo? |
| {*N*} | match *N* occurrences of preceding RE | \d{3} |
| {*M*,*N*} | match from *M* to *N* occurrences of preceding RE | \d{5,9} |
| […] | match any single character from *character* class | [aeiou] |
| [..*x*-*y*..] | match any single character in the *range from* x to y | [0-9], [A-Za-z] |
| [^…] | *do not match* any character from character class, including any ranges, if present | [^aeiou], [^A-Za-z0-9_] |
| (*\|+\|?\|{})? | apply non-greedy versions of above occurrence/repetition symbols ( *, +, ?, {}) | .*?\w |
| (…) | match enclosed RE and save as *subgroup* | (\d{3})?, f(oo\|u)bar |
| ***Special Characters*** | | |
| \d | match any decimal *digit,* same as [0-9] (\D is inverse of \d: do not match any numeric digit) | data\d+.txt |
| \w | match any *alphanumeric* character, same as [A-Za-z0-9_] (\W is inverse of \w) | [A-Za-z_]\w+ |
| \s | match *any whitespace* character, same as [ \n\t\r\v\f] (\S is inverse of \s) | of\sthe |
| \b | match any *word boundary* (\B is inverse of \b) | \bThe\b |

| | | |
|---|---|---|
| \nn | match saved *subgroup* nn (see (…) above) | price: \16 |
| \c | match any *special character* c verbatim (i.e., without its special meaning, literal) | \., \\, * |
| \A (\Z) | match *start (end) of string* (also see ^ and $ above) | \ADear |

## Matching more than one RE pattern with alternation ( | )

The pipe symbol ( | ), a vertical bar on your keyboard, indicates an *alternation* operation, meaning that it is used to choose from one of the different regular expressions which are separated by the pipe symbol. For example, below are some patterns which employ alternation, along with the strings they match:

| RE Pattern | Strings Matched |
|---|---|
| at\|home | at, home |
| r2d2\|c3po | r2d2, c3po |
| bat\|bet\|bit | bat, bet, bit |

With this one symbol, we have just increased the flexibility of our regular expressions, enabling the matching of more than just one string. Alternation is also sometimes called union or logical OR.

## Matching any single character ( . )

The dot or period ( . ) symbol matches any single character except for NEWLINE (Python REs have a compilation flag [S or DOTALL] which can override this to include NEWLINEs.). Whether letter, number, whitespace not including "\n," printable, non-printable, or a symbol, the dot can match them all.

| RE Pattern | Strings Matched |
|---|---|
| f.o | any character between "f" and "o," e.g., fao, f9o, f#o, etc. |
| .. | any pair of characters |
| .end | any character before the string end |

**Q:**

"What if I want to match the dot or period character?"

**A:**

In order to specify a dot character explicitly, you must escape its functionality with a backslash, as in "\."

## Matching from the beginning or end of strings or word boundaries ( ^/$)

There are also symbols and related special characters to specify searching for patterns at the beginning and ending of strings. To match a pattern starting from the beginning, you must use the carat symbol ( ^ ) or the special character \A (backslash-capital "A"). The latter is primarily for keyboards which do not have the carat symbol, i.e., international. Similarly, the dollar sign ( $ ) or \Z will match a pattern from the end of a string.

Patterns which use these symbols differ from most of the others we describe in this chapter since they dictate location or position. In the Core Note above, we noted that a distinction is made between "matching," attempting matches of entire strings starting at the beginning, and "searching," attempting matches from anywhere within a string. Because we are looking specifically at symbols and special characters which deal with position, they make sense only when applied to searching.

That said, here are some examples of "edge-bound" RE search patterns:

| RE Pattern | Strings Matched |
|---|---|
| ^From | any string which starts with From |
| /bin/tcsh$ | any string which ends with /bin/tcsh |
| ^Subject: hi$ | any string consisting solely of the string Subject: hi |

Again, if you want to match either (or both) of these characters verbatim, you must use an escaping backslash. For example, if you wanted to match any string which ended with a dollar sign, one possible RE solution would be the pattern ".*\$$".

The \b and \B special characters will match the empty string, meaning that they can start performing the match anywhere. The difference is that \b will match a pattern to a word boundary, meaning that a pattern must be at the beginning of a word, whether there are any characters in front of it (word in the middle of a string) or not (word at the beginning of a line). And likewise, \B will match a pattern only if it appears starting in the middle of a word (i.e., not at a word boundary). Here are some examples:

| RE Pattern | Strings Matched |
|---|---|
| the | any string containing the |
| \bthe | any word which starts with the |
| \bthe\b | matches only the word the |
| \Bthe | any string which contains but does not begin with the |

## Creating character classes ( [ ] )

While the dot is good for allowing matches of any symbols, there may be occasions where there are specific characters you want to match. For this reason, the bracket

symbols ( `[ ]` ) were invented. The regular expression will match from any of the enclosed characters. Here are some examples:

| RE Pattern | Strings Matched |
|---|---|
| `b[aeiu]t` | `bat, bet, bit, but` |
| `[cr][23][dp][o2]` | "r" or "c" then "2" or "3" followed by "d" or "p" and finally, either "o" or "2," e.g., `c2do, r3p2, r2d2, c3po,` etc. |

One side note regarding the RE "`[cr][23][dp][o2]`"—a more restrictive version of this RE would be required to allow only "r2d2" or "c3po" as valid strings. Because brackets merely imply "logical OR" functionality, it is not possible to use brackets to enforce such a requirement. The only solution is to use the pipe, as in "`r2d2|c3po`".

For single character REs, though, the pipe and brackets are equivalent. For example, let's start with the regular expression "ab" which matches only the string with an "a" followed by a "b." If we wanted either a one-letter string, i.e., either "a" or a "b," we could use the RE "`[ab]`". Because "a" and "b" are individual strings, we can also choose the RE "`a|b`". However, if we wanted to match the string with the pattern "ab" followed by "cd," we cannot use the brackets because they work only for single characters. In this case, the only solution is "`ab|cd,`" similar to the "`r2d2/c3po`" problem just mentioned.

## Denoting ranges ( `-` ) and negation ( `^` )

In addition to single characters, the brackets also support ranges of characters. A hyphen between a pair of symbols enclosed in brackets is used to indicate a range of characters, e.g., A–Z, a–z, or 0–9 for uppercase letters, lowercase letters, and numeric digits, respectively. This is a lexicographic range, so you are not restricted to using just alphanumeric characters. Additionally, if a caret ( `^` ) is the first character immediately inside the open left bracket, this symbolizes a directive to not match any of the characters in the given character set.

| RE Pattern | Strings Matched |
|---|---|
| `z.[0-9]` | "z" followed by any character then followed by a single digit |
| `[r-u][env-y][us]` | "r" "s," "t" or "u" followed by "e," "n," "v," "w," "x," or "y" followed by "u" or "s" |
| `[^aeiou]*` | zero or more (*symbol introduced in next subsection) non-vowels (EXERCISE: Why do we say "non-vowels" rather than "consonants?") |
| `[^\t\n]+` | one or more (+symbol introduced in next subsection) characters up to, but not including, the first TAB or NEWLINE encountered |
| `["-a]` | in an ASCII system, all characters which fall between """ and "a," i.e., between ordinals 34 and 97. |

## Multiple occurrence/repetition using closure operators ( `*, +, ?, { }` )

We will now introduce the most common RE notations, namely, the special symbols `*`, `+`, and `?`, all of which can be used to match single, multiple, or no occurrences of string patterns. The asterisk or star operator ( `*` ) will match zero or more occurrences of the RE immediately to its left (in language and compiler theory, this operation is known as the Kleene Closure). The plus operator ( `+` ) will match one or more occurrences of an RE (known as Positive Closure), and the question mark operator ( `?` ) will match exactly 0 or 1 occurrences of an RE.

There are also brace operators ( `{ }` ) with either a single value or a comma-separated pair of values. These indicate a match of exactly *N* occurrences (for `{ N }`) or a range of occurrences, i.e., `{M, N}` will match from *M* to *N* occurrences. These symbols may also be escaped with the backslash, i.e., "`\*`" matches the asterisk, etc.

Finally, the question mark ( `?` ) is overloaded so that if it follows any of the following symbols, it will direct the regular expression engine to match as few repetitions as possible.

Here are some examples using the closure operators:

| RE Pattern | Strings Matched |
|---|---|
| `[dn]ot?` | "d" or "n," followed by an "o" and, at most, one "t" after that, i.e., `do, no, dot, not` |
| `0?[1-9]` | any numeric digit, possibly prepended with a "0," e.g., the set of numeric representations of the months January to September, whether single- or double-digits |
| `[0-9]{15,16}` | fifteen or sixteen digits, e.g., credit card numbers |
| `</?[^>]+>` | strings which match all valid (and invalid) HTML tags |
| `[KQRBNP][a-h][1-8]-[a-h][1-8]` | Legal chess move in "long algebraic" notation (move only, no capture, check, etc.), i.e., strings which start with any of "K," "Q," "R," "B," "N," or "P" followed by a hyphenated-pair of chess board grid locations from "a1" to "h8" (and everything in between), with the first coordinate indicating the former position and the second being the new position. |

## Special characters representing character sets

We also mentioned that there are special characters which may represent character sets. Rather than using a range of "0–9," you may simply use "`\d`" to indicate the match of any decimal digit. Another special character "`\w`" can be used to denote the entire alphanumeric character class, serving as a shortcut for "`A-Za-z0-9_`," and "`\s`" for whitespace characters. Uppercase versions of these strings symbolizes a *non*-match, i.e., "`\D`" matches any non-decimal digit (same as "`[^0-9]`"), etc.

Using these shortcuts, we will present a few more complex examples:

| RE Pattern | Strings Matched |
|---|---|
| `\w+-\d+` | alphanumeric string and number separated by a hyphen |

| | |
|---|---|
| [A-Za-z]\w* | alphabetic first character, additional characters (if present) can be alphanumeric (almost equivalent to the set of valid Python identifiers [see exercises]) |
| \d{3}-\d{3}-\d{4} | (American) telephone numbers with an area code prefix, as in 800-555-1212 |
| \w+@\w+\.com | simple e-mail addresses of the form *XXX@YYY.com* |

Note that all special characters, including all the ones mentioned before such as "\A," "\B," "\d," etc., may or may not have ASCII equivalents. To be sure you are using the regular expression versions, it would be a safe bet to use raw strings to escape backslash functionality (see the Core Note later in this chapter).

Also, the "\w" and "\W" alphanumeric character sets are affected by the L or LOCALE compilation flag and in Python 1.6 and newer, by Unicode flags.

## Designating groups with parentheses ( ( ) )

Now, perhaps we have achieved the goal of matching a string and discarding non-matches, but in some cases, we may also be more interested in the data that we did match. Not only do we want to know whether the entire string matched our criteria, but whether we can also extract any specific strings or substrings which were part of a successful match. The answer is yes. To accomplish this, surround any RE with a pair of parentheses.

A pair of parentheses ( ( ) ) can accomplish either (or both) of the below when used with regular expressions:

grouping regular expressions

matching subgroups

One good example for wanting to group regular expressions is when you have two different REs with which you want to compare a string. Another reason is to group an RE in order to use a repetition operator on the entire RE (as opposed to an individual characters or character classes).

One side-effect of using parentheses is that the substring which matched the pattern is saved for future use. These subgroups can be recalled for the same match or search, or extracted for post-processing. Why are matches of subgroups important? The main reason is that there are times where you want to extract the patterns you match, in addition to making a match.

For example, what if we decided to match the pattern "\w+-\d+" but wanted save the alphabetic first part and the numeric second part individually? This may be desired because with any successful match, we may want to see just what those strings were that matched our RE patterns. If we add parentheses to both subpatterns, i.e., "(\w+)-(\d+)," then we can access each of the matched subgroups individually. Subgrouping is preferred because the alternative is to write code to determine we have a match, then

execute another separate routine (which we also had to create) to parse the entire match just to extract both parts. Why not let Python do it, since it is a supported feature of the `re` module, instead of "reinventing the wheel"?

| RE Pattern | Strings Matched |
|---|---|
| `\d+(\.\d*)?` | strings representing simple floating point number, that is, any number of digits followed optionally by a single decimal point and zero or more numeric digits, as in "0.004," "2," "75.," etc. |
| `(Mr?s?\. )?[A-Z][a-z]* [ A-Za-z-]+` | first name and last name, with a restricted first name (must start with uppercase; lowercase only for remaining letters, if any), the full name prepended by an optional title of "Mr.," "Mrs.," "Ms.," or "M.," and a flexible last name, allowing for multiple words, dashes, and uppercase letters |

## REs and Python

Now that we know all about regular expressions, we can examine how Python currently supports regular expressions through the `re` module. The re module was introduced to Python in version 1.5. If you are using an older version of Python, you will have to use the now-obsolete `regex` and `regsub` modules—these older modules are more Emacs-flavored, are not as full-featured, and are in many ways incompatible with the current `re` module.

However, regular expressions are still regular expressions, so most of the basic concepts from this section can be used with the old `regex` and `regsub` software. In contrast, the new re module supports the more powerful and regular Perl-style (Perl5) REs, allows multiple threads to share the same compiled RE objects, and supports named subgroups. In addition, there is a transition module called `reconvert` to help developers move from `regex/regsub` to `re.` However, be aware that although there are different flavors of regular expressions, we will primarily focus on the current incarnation for Python.

The `re` engine was rewritten in 1.6 for performance enhancements as well as adding Unicode support. The interface was not changed, hence the reason the module name was left alone. The new `re` engine—known internally as `sre` —thus replaces the existing 1.5 engine—internally called `pcre.`

### `re` Module: Core Functions and Methods

The chart in Table 15.2 lists the more popular functions and methods from the `re` module. Many of these functions are also available as methods of compiled regular expression objects "regex objects" and RE "match objects." In this subsection, we will look at the two main functions/methods, `match()` and `search(),` as well as the `compile()` function. We will introduce several more in the next section, but for more information on all these and the others which we do not cover, we refer you to the Python documentation.

| Table 15.2. Common Regular Expression Functions and Methods | |
|---|---|
| ***Function/Method*** | ***Description*** |

| re ***Module Function Only*** | |
|---|---|
| compile(*pattern*, *flags*=0) | compile RE *pattern* with any optional *flags* and return a regex object |
| re ***Module Functions and regex Object Methods*** | |
| match(*pattern*, *string*, *flags*=0) | attempt to match RE *pattern* to *string* with optional *flags*; return match object on success, None on failure |
| search(*pattern*, *string*, *flags*=0) | search for first occurrence of RE *pattern* within *string* with optional *flags*; return match object on success, None on failure |
| findall(*pattern*, *string*) | look for all (non-overlapping) occurrences of *pattern* in *string*; return a list of matches (new as of Python 1.5.2) |
| split(*pattern*, *string*, *max*=0) | split *string* into a list according to RE *pattern* delimiter and return list of successful matches, splitting at most *max* times (split all occurrences is the default) |
| sub(*pattern*, *repl*, *string*, *max*=0) | replace all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided (also see subn() which, in addition, returns the number of substitutions made) |
| ***Match Object Methods*** | |
| group(*num*=0) | return entire match (or specific subgroup *num*) |
| groups() | return all matching subgroups in a tuple (empty if there weren't any) |

**NOTE**

In the previous chapter, we described how Python code is eventually compiled into bytecode which is then executed by the interpreter. In particular, we mentioned that calling eval() or **exec** with a code object rather than a string provides a significant performance improvement due to the fact that the compilation process does not have to be performed. In other words, using precompiled code objects is faster than using strings because the interpreter will have to compile it into a code object (anyway) before execution.

The same concept applies to REs—regular expression patterns must be compiled into regex objects before any pattern matching can occur. For REs which are compared many times during the course of execution, we highly recommend using precompilation first because, again, REs have to be compiled anyway, so doing it ahead of time is prudent for performance reasons. re.compile() provides this functionality.

The module functions do cache the compiled objects, though, so it's not as if every search() and match() with the same RE pattern requires compilation. Still, you save the cache lookups and do not have to make function calls with the same string over and over. In Python 1.5.2, this cache held up to 20 compiled RE objects, but in 1.6, due to the

additional overhead of Unicode awareness, the compilation engine is a bit slower, so the cache has been extended to 100 compiled regex objects.

### Compiling REs with `compile()`

Almost all of the `re` module functions we will be describing shortly are available as methods for regex objects. Remember, even with our recommendation, precompilation is not required. If you compile, you will use methods; if you don't, you will just use functions. The good news is that either way, the names are the same whether a function or a method. (This is the reason why there are module functions and methods which are identical, i.e., `search(), match(),` etc., in case you were wondering.) Since it saves one small step for most of our examples, we will use strings instead. We will throw in a few with compilation though just so you know how it is done.

Optional flags may be given as arguments for specialized compilation. These flags allow for case-insensitive matching, using system locale settings for matching alphanumeric characters, etc. Please refer to the documentation for more details. These flags, some of which have been briefly mentioned (i.e. `DOTALL, LOCALE`), may also be given to the module versions of `match()` and `search()` for a specific pattern match attempt—these flags are mostly for compilation reasons, hence the reason why they can be passed to the module versions of `match()` and `search()` which do compile an RE pattern once. If you want to use these flags with the methods, they must already be integrated into the compiled regex objects.

In addition to the methods below, regex objects also have some data attributes, two of which include any compilation flags given as well as the regular expression pattern compiled.

### Match objects and the `group()` and `groups()` Methods

There is another object type in addition to the regex object when dealing with regular expressions, the *match object.* These objects are those which are returned on successful calls to `match()` or `search()`. Match objects have two primary methods, `group()` and `groups()`.

`group()` will either return the entire match, or a specific subgroup, if requested. `groups()` will simply return a tuple consisting of only/all the subgroups. If there are no subgroups requested, then `groups()` returns an empty tuple while `group()` still returns the entire match.

Python REs also allow for named matches, which are beyond the scope of this introductory section on REs. We refer you to the complete `re` module documentation regarding all the more advanced details we have omitted here.

### Matching strings with `match()`

`match()` is the first `re` module function and RE object (regex object) method we will look at. The `match()` function attempts to match the pattern to the string, starting at the beginning. If the match is successful, a match object is returned, but on failure, `None` is returned. The `group()` method of a match object can be used to show the successful match. Here is an example of how to use `match()` [and `group()`]:

```
>>> m = re.match('foo', 'foo') # pattern matches string
>>> if m != None:               # show match if successful
…     m.group()
…
`foo'
```

The pattern "foo" matches exactly the string "foo." We can also confirm that m is an example of a match object from within the interactive interpreter:

```
>>> m                        # confirm match object returned
<re.MatchObject instance at 80ebf48>
```

Here is an example of a failed match where None is returned:

```
>>> m = re.match('foo', 'bar')# pattern does not match string
>>> if m != None: m.group()   # (1-line version of if clause)
…
>>>
```

The match above fails, thus None is assigned to `m`, and no action is taken due to the way we constructed our **`if`** statement. For the remaining examples, we will try to leave out the **`if`** check for brevity, if possible, but in practice it is a good idea to have it there to prevent `AttributeError` exceptions (`None` is returned on failures, which does not have a `group()` attribute [method].)

A match will still succeed even if the string is longer than the pattern as long as the pattern matches from the beginning of the string. For example, the pattern "foo" will find a match in the string "food on the table" because it matches the pattern from the beginning:

```
>>> m = re.match('foo', 'food on the table')# match succeeds
>>> m.group()
'foo'
```

As you can see, although the string is longer than the pattern, a successful match was made from the beginning of the string. The substring "foo" represents the match which was extracted from the larger string.

We can even sometimes bypass saving the result altogether, taking advantage of Python's object-oriented nature:

```
>>> re.match('foo', 'food on the table').group()
'foo'
```

Note from a few paragraphs above that an `AttributeError` will be generated on a non-match.

### Looking for a pattern within a string with `search()` (searching vs. matching)

The chances are greater that the pattern you seek is somewhere in the middle of a string, rather than at the beginning. This is where `search()` comes in handy. It works exactly in the same way as match except that it searches for the first occurrence of the given RE pattern anywhere with its string argument. Again, a match object is returned on success and `None` otherwise.

We will now illustrate the difference between `match()` and `search()`. Let us try a longer string match attempt. This time, we will try to match our string "foo" to "seafood:"

```
>>> m = re.match('foo', 'seafood')        # no match
>>> if m != None: m.group()
…
>>>
```

As you can see, there is no match here. `match()` attempts to match the pattern to the string from the beginning, i.e., the "f" in the pattern is matched against the "s" in the string, which fails immediately. However, the string "foo" *does* appear (elsewhere) in "seafood," so how do we get Python to say "yes?" The answer is by using the `search()` function. Rather than attempting a *match,* `search()` looks for the first occurrence of the pattern within the string. `search()` searches strictly from left to right.

```
>>> m = re.search('foo', 'seafood')      # use search() instead
>>> if m != None: m.group()
…
'foo'                    # search succeeds where match failed
>>>
```

We will be using the `match()` and `search()` regex object methods and the `group()` and `groups()` match object methods for the remainder of this subsection, exhibiting a broad range of examples of how to use regular expressions with Python. We will be using almost all of the special characters and symbols which are part of the regular expression syntax.

### Matching more than one string ( | )

In Section 15.2, we used the pipe in the RE "`bat|bet|bit`". Here is how we would use that RE with Python:

```
>>> bt = 'bat|bet|bit'              # RE pattern: bat, bet, bit
>>> m = re.match(bt, 'bat')         # 'bat' is a match
>>> if m != None: m.group()
…
'bat'
>>> m = re.match(bt, 'blt')         # no match for 'blt'
>>> if m != None: m.group()
…
>>> m = re.match(bt, 'He bit me!') # does not match string
>>> if m != None: m.group()
…
>>> m = re.search(bt, 'He bit me!')# found \qbit\q via search
>>> if m != None: m.group()
…
'bit'
```

### Matching any single character ( . )

In the examples below, we show that a dot cannot match a NEWLINE or a non-character, i.e., the empty string:

```
>>> anyend = '.end'
>>> m = re.match(anyend, 'bend')      # dot matches 'b'
>>> if m != None: m.group()
…
'bend'
>>> m = re.match(anyend, 'end')       # no char to match
>>> if m != None: m.group()
…
>>> m = re.match(anyend, '\nend')     # any char except \n
>>> if m != None: m.group()
…
>>> m = re.search('.end', 'The end.') # matches ' ' in search
>>> if m != None: m.group()
…
```

```
' end'
```

Below is an example of searching for a real dot (decimal point) in a regular expression where we escape its functionality with a backslash:

```
>>> patt314 = '3.14'                    # RE dot
>>> pi_patt = '3\.14'                   # literal dot (dec. point)
>>> m = re.match(pi_patt, '3.14')       # exact match
>>> if m != None: m.group()
…
'3.14'
>>> m = re.match(patt314, '3014')       # dot matches '0'
>>> if m != None: m.group()
…
'3014'
>>> m = re.match(patt314, '3.14')       # dot matches '.'
>>> if m != None: m.group()
…
'3.14'
```

### Creating character classes ( [ ] )

Earlier, we had a long discussion regarding "`[cr][23][dp][o2]`" and how it differs from "`r2d2|c3po`". With the examples below, we will show that "`r2d2|c3po`" is more restrictive than "`[cr][23][dp][o2]`":

```
>>> m = re.match('[cr][23][dp][o2]', 'c3po')# matches \qc3po\q
>>> if m != None: m.group()
…
'c3po'
>>> m = re.match('[cr][23][dp][o2]', 'c2do')# matches 'c2do'
>>> if m != None: m.group()
…
'c2do'
>>> m = re.match('r2d2|c3po', 'c2do')# does not match 'c2do'
>>> if m != None: m.group()
…
>>> m = re.match('r2d2|c3po', 'r2d2')# matches 'r2d2'
>>> if m != None: m.group()
…
'r2d2'
```

### Repetition, special characters, and grouping

The most common aspects of REs involve the use of special characters, multiple occurrences of RE patterns, and using parentheses to group and extract submatch patterns.

One particular RE we looked at related to simple e-mail addresses ("`\w+@\w+\.com`"). Perhaps we want to match more e-mail addresses than this RE allows. In order to support an additional hostname in front of the domain, i.e., "`www.xxx.com`" as opposed to accepting only "`xxx.com`" as the entire domain, we have to modify our existing RE. To indicate that the hostname is optional, we create a pattern which matches the hostname (followed by a dot), use the ? operator indicating zero or one copy of this pattern, and insert the optional RE into our previous RE as follows: "`\w+@(\w+\.)?\w+\.com`". As you can see from the examples below, either one or two names are now accepted in front of the "`.com`".

```
>>> patt = '\w+@(\w+\.)?\w+\.com'
>>> re.match(patt, 'nobody@xxx.com').group()
'nobody@xxx.com'
>>> re.match(patt, 'nobody@www.xxx.com').group()
'nobody@www.xxx.com'
```

Furthermore, we can even extend our example to allow any number of intermediate subdomain names with the following pattern: "`\w+@(\w+\.)*\w+\.com`":

```
>>> patt = '\w+@(\w+\.)*\w+\.com'
>>> re.match(patt,
'nobody@www.xxx.yyy.zzz.com').group()
'nobody@www.xxx.yyy.zzz.com'
```

However, we must add the disclaimer that using solely alphanumeric characters does not match all the possible characters which may make up e-mail addresses. The above RE patterns would not match a domain such as "`xxx-yyy.com`" or other domains with "`\W`" characters.

Earlier, we discussed the merits of using parentheses to match and save subgroups for further processing rather than coding a separate routine to manually parse a string after an RE match had been determined. In particular, we discussed a simple RE pattern of an alphanumeric string and a number separated by a hyphen, "`\w+-\d+,`" and how adding subgrouping to form a new RE, "`(\w+)-(\d+),`" would do the job. Here is how the original RE works:

```
>>> m = re.match('\w\w\w-\d\d\d', 'abc-123')
>>> if m != None: m.group()
…
'abc-123'

>>> m = re.match('\w\w\w-\d\d\d', 'abc-xyz')
>>> if m != None: m.group()
…
```

```
>>>
```

In the above code, we created an RE to recognize three alphanumeric characters followed by three digits. Testing this RE on "`abc-123,`" we obtained with positive results while "`abc-xyz`" fails. We will now modify our RE as discussed before to be able to extract the alphanumeric string and number. Note how we can now use the `group()` method to access individual subgroups or the `groups()` method to obtain a tuple of all the subgroups matched:

```
>>> m = re.match('(\w\w\w)-(\d\d\d)', 'abc-123')
>>> m.group()                   # entire match
'abc-123'
>>> m.group(1)                  # subgroup 1
'abc'
>>> m.group(2)                  # subgroup 2
'123'
>>> m.groups()                  # all subgroups
('abc', '123')
```

As you can see, `group()` is used in the normal way to show the entire match, but can also be used to grab individual subgroup matches. We can also use the `groups()` method to obtain a tuple of all the substring matches.

Here is a simpler example showing different group permutations, which will hopefully make things even more clear:

```
>>> m = re.match('ab', 'ab')       # no subgroups
>>> m.group()                      # entire match
'ab'
>>> m.groups()                     # all subgroups
()
>>>
>>> m = re.match('(ab)', 'ab')     # one subgroup
>>> m.group()                      # entire match
'ab'
>>> m.group(1)                     # subgroup 1
'ab'
>>> m.groups()                     # all subgroups
('ab',)
>>>
>>> m = re.match('(a)(b)', 'ab')   # two subgroups
>>> m.group()                      # entire match
'ab'
>>> m.group(1)                     # subgroup 1
'a'
>>> m.group(2)                     # subgroup 2
'b'
>>> m.groups()                     # all subgroups
```

```
('a', 'b')
>>>
>>> m = re.match('(a(b))', 'ab')  # two subgroups
>>> m.group()                     # entire match
'ab'
>>> m.group(1)                    # subgroup 1
'ab'
>>> m.group(2)                    # subgroup 2
'b'
>>> m.groups()                    # all subgroups
('ab', 'b')
```

#### Matching from the beginning and end of strings and on word boundaries

The following examples highlight the positional RE operators. These apply more for searching than matching because `match()` always starts at the beginning of a string.

```
>>> m = re.search('^The', 'The end.')      # match
>>> if m != None: m.group()
…
'The'
>>> m = re.search('^The', 'end. The')      # not at beginning
>>> if m != None: m.group()
…
>>> m = re.search(r'\bthe', 'bite the dog')# at a boundary
>>> if m != None: m.group()
…
'the'
>>> m = re.search(r'\bthe', 'bitethe dog') # no boundary
>>> if m != None: m.group()
…
>>> m = re.search(r'\Bthe', 'bitethe dog') # no boundary
>>> if m != None: m.group()
…
'the'
```

You will notice the appearance of raw strings here. You may want to take a look at the Core Note towards the end of the chapter for clarification on why they are here. In general, it is a good idea to use raw strings with regular expressions.

### Other `re` Module Functions and Methods

There are four other `re` module functions and regex object methods which we think you should be aware of: `findall()`, `sub()`, `subn()`, and `split()`.

#### Finding every occurrence with `findall()`

`findall()` is new to Python as of version 1.5.2. It looks for all non-overlapping occurrences of an RE pattern in a string. It is similar to `search()` in that it performs a string search, but it differs from `match()` and `search()` in that `findall()` always returns a list. The list will be empty if no occurrences are found but if successful, it will consist of all matches found (grouped in left-to-right order of occurrence).

```
>>> re.findall('car', 'car')
['car']
>>> re.findall('car', 'scary')
['car']
>>> re.findall('car', 'carry the barcardi to the car')
['car', 'car', 'car']
```

*Subgroup* searches result in a more complex list returned, and that makes sense, because subgroups are a mechanism which will allow you to extract specific patterns from within your single regular expression, such as matching an area code which is part of a complete telephone number, or a login name which is part of an entire e-mail address.

For a single successful match, each subgroup match is a single element of the resulting list returned by `findall();` for multiple successful matches, each subgroup match is a single element in a tuple, and such tuples (one for each successful match) are the elements of the resulting list. This part may sound confusing at first, but if you try different examples, it will help clarify things.

### Searching and replacing with `sub()` [and `subn()`]

There are two functions/methods for search-and-replace functionality: `sub()` and `subn()`. They are both almost identical and replace all matched occurrences of the RE pattern in a string with some sort of replacement. The replacement is usually a string, but it can also be a function which returns a replacement string. `subn()` is exactly the same as `sub()`, but it also returns the total number of substitutions made—both the newly-substituted string and the substitution count are returned as a 2-tuple.

```
>>> re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
'attn: Mr. Smith\012\012Dear Mr. Smith,\012'
>>>
>>> re.subn('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
('attn: Mr. Smith\012\012Dear Mr. Smith,\012', 2)
>>>
>>> print re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
attn: Mr. Smith

Dear Mr. Smith,

>>> re.sub('[ae]', 'X', 'abcdef')
'XbcdXf'
>>> re.subn('[ae]', 'X' , 'abcdef')
```

```
('XbcdXf', 2)
```

### Splitting (on delimiting pattern) with `split()`

The `re` module and RE object method `split()` work similar to its string counterpart, but rather than splitting on a fixed string, it splits a string based on an RE pattern, adding some significant power to string splitting capabilities. If you do not want the string split for every occurrence of the pattern, you can specify the maximum number of splits by setting a value (other than zero) to the `max` argument.

If the delimiter given is not a regular expression which uses special symbols to match multiple patterns, then `re.split()` works in exactly the same manner as `string.split()`, as illustrated in the example below (which splits on a single colon):

```
>>> re.split(':', 'str1:str2:str3')
['str1', 'str2', 'str3']
```

But with regular expressions involved, we have an even more powerful tool. Take, for example, the output from the Unix who command, which lists all the users logged into a system:

```
% who
wesc       console      Jun 20 20:33
wesc       pts/9        Jun 22 01:38    (192.168.0.6)
wesc       pts/1        Jun 20 20:33    (:0.0)
wesc       pts/2        Jun 20 20:33    (:0.0)
wesc       pts/4        Jun 20 20:33    (:0.0)
wesc       pts/3        Jun 20 20:33    (:0.0)
wesc       pts/5        Jun 20 20:33    (:0.0)
wesc       pts/6        Jun 20 20:33    (:0.0)
wesc       pts/7        Jun 20 20:33    (:0.0)
wesc       pts/8        Jun 20 20:33    (:0.0)
```

Perhaps we want to save some user login information such as login name, teletype they logged in at, when they logged in, and from where. Using `string.split()` on the above would not be effective, since the spacing is erratic and inconsistent. The other problem is that there is a space between the month, day, and time for the login timestamps. We would probably want to keep these fields together.

You need some way to describe a pattern such as, "split on two or more spaces." This is easily done with regular expressions. In no time, we whip up the RE pattern "\s\s+," which does mean at least two whitespace characters. Let's create a program called

rewho.py that reads the output of the who command, presumably saved into a file called whodata.txt. Our rewho.pyscript initially looks something like this:

```
import re
f = open('whodata.txt', 'r')
for eachLine in f.readlines():
    print re.split('\s\s+', eachLine)
f.close()
```

We will now execute the who command, saving the output into whodata.txt, and then call rewho.py and take a look at the results:

```
% who > whodata.txt
% rewho.py
['wesc', 'console', 'Jun 20 20:33\012']
['wesc', 'pts/9', 'Jun 22 01:38\011(192.168.0.6)\012']
['wesc', 'pts/1', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/2', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/4', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/3', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/5', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/6', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/7', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/8', 'Jun 20 20:33\011(:0.0)\012']
```

It was a good first try, but not quite correct. For one thing, we did not anticipate a single TAB (ASCII \011) as part of the output (which looked like at least 2 spaces, right?), and perhaps we aren't really keen on saving the NEWLINE (ASCII \012) which terminates each line. We are now going to fix those problems as well as improve the overall quality of our application by making a few more changes.

First, we would rather run the who command from within the script, instead of doing it externally and saving the output to a whodata.txt file—doing this repeatedly gets tiring rather quickly. To accomplish invoking another program from within ours, we call upon the os.popen() command, discussed briefly in Section 14.5.2. Although os.popen() is available only on Unix systems, the point is to illustrate the functionality of re.split(), which is available on all platforms.

We shall also employ the map() built-in function along with string.strip() to get rid of the trailing NEWLINEs. Finally, we will add the detection of a single TAB as an additional, alternative re.split() delimiter by adding it to the regular expression. Presented below in Example 15.1. is the final version of our rewho.pyscript:

**Example 15.1. Split Output of Unix who Command (rewho.py)**

This script calls the **who** command and parses the input by splitting up its data along various types of whitespace characters.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from os import popen
004 4  from re import split
005 5  from string import strip
006 6
007 7  f = popen('who', 'r')
008 8  for eachLine in map(strip, f.readlines()):
009 9      print split('\s\s+|\t', eachLine)
010 10 f.close()
011  <$nopage>
```

Running this script, we now get the following (correct) output:

```
% rewho.py
['wesc', 'console', 'Jun 20 20:33']
['wesc', 'pts/9', 'Jun 22 01:38', '(192.168.0.6)']
['wesc', 'pts/1', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/2', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/4', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/3', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/5', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/6', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/7', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/8', 'Jun 20 20:33', '(:0.0)']
```

A similar exercise can be achieved in a DOS/Windows environment using the `dir` command in place of `who`.

**NOTE**

You may have seen the use of raw strings in some of the examples above. Regular expressions were a strong motivation for the advent of raw strings. The reason is because of conflicts between ASCII characters and regular expression special characters. As a special symbol, "`\b`" represents the ASCII character for backspace, but "`\b`" is also a regular expression special symbol, meaning "match" on a word boundary. In order for the RE compiler to not interpret a "`\b`" in your string as a backspace, you need to escape it using the backslash, resulting in "`\\b`."

This can get messy, especially if you have a lot of special characters in your string, which adds to the confusion. We were introduced to raw strings back in *Section 6.4.2*, and they can be (and are often) used to help keep REs looking somewhat manageable. In fact, many Python programmers swear by these and only use raw strings when defining regular expressions.

Here are some examples of differentiating between the backspace "\b" and the regular expression "\b," with and without raw strings:

```
>>> m = re.match('\bblow', 'blow')     # backspace, no match
>>> if m != None: m.group()
…
>>> m = re.match('\\bblow', 'blow')    # escaped \, now it works
>>> if m != None: m.group()
…
'blow'
>>> m = re.match(r'\bblow', 'blow\q)  # or use raw string
instead
>>> if m != None: m.group()
…
'blow'
```

You may have recalled that we had no trouble using "\d" in our regular expressions without using raw strings. That is because there is no ASCII equivalent special character, so the regular expression compiler already knew you meant a decimal digit.

## Regular Expression Adventures

We will now run through an in-depth example of the different ways of using regular expressions for string manipulation. The first step is to come up with some code that actually generates some random (but-not-so-random) data on which to operate. In Example 15.2, we present gendata.py, a script which generates a data set. Although this program simply displays the generated set of strings to standard output, this output may very well be redirected to a test file.

**NOTE**

Unix systems, as well as others, use architecture-size integers to represent the current time in seconds. Since most systems today are 32-bit, the total amount of time recognized by any platform using this mechanism is $2^{32}$ seconds. Such integers are signed, so we really only have $2^{31}$-1 seconds.

The current time is recognized as the number of seconds which have elapsed since time zero, which is pegged at midnight, January 1, 1970. Moving forward to the maximum possible positive 32-bit signed integer ($2^{31}$ - 1), we arrive at the "end of time," which evaluates to Tuesday morning, January 19, 2038 at 3:14 AM and 7 seconds using Universal Coordinated Time (UTC/GMT). Hopefully by then, we would have discontinued the use of 32-bit systems. This phenomena is otherwise known as the Y2038 problem.)

Here is one way you could find out what the special date/time it is for your local time, using Python:

```
>>> import sys, time
>>> time.asctime(time.localtime(sys.maxint))# Pacific Time
'Mon Jan 18 19:14:07 2038'
```

`sys.maxint` has the last possible second using a 32-bit integer. We feed that time in seconds to `time.localtime()` to obtain the tuple for your/our local time (here we are on Pacific Time), and finally, we ship that tuple off to `time.asctime()` to obtain the standard timestamp for the last possible second. As you can see from our example, we are eight hours west of the Prime/Greenwich Meridian.

This is not as much a Python Core Note as it is a general programming note, but should be nevertheless discussed for common knowledge since it applies to all 32-bit systems with applications using on the C language, regardless of platform, i.e., UNIX and non-UNIX, which use UNIX-style dating. In the gendata.py script coming up, we randomly generate integers, effectively generating random dates for our application.

This script generates strings with three fields, delimited by a pair of colons, or a double-colon. The first field is a random (32-bit) integer, which is converted to a date (see the accompanying Core Note). The next field is a randomly-generated electronic mail (e-mail) address, and the final field is a set of integers separated by a single dash ( - ).

**Example 15.2. Data Generator for RE Exercises (`gendata.py`)**

Create random data for regular expressions practice and output the generated data to the screen.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from random import randint,choice
004 4  from string import lowercase
005 5  from sys import maxint
006 6  from time import ctime
007 7
008 8  doms = ( 'com', 'edu', 'net', 'org', 'gov' )
009 9
010 10  for i in range(randint(5, 10)):
011 11     dtint = randint(0, maxint-1)          # pick date
012 12     dtstr = ctime(dtint)                  # date string
013 13
014 14     shorter = randint(4, 7)               # login shorter
015 15     em = ''
016 16     for j in range(shorter):             # generate login
017 17        em = em + choice(lowercase)
018 18
019 19    longer = randint(shorter, 12)          # domain longer
```

```
020 20    dn = ''
021 21    for j in range(longer):                # create domain
022 22        dn = dn + choice(lowercase)
023 23
024 24    print '%s::%s@%s.%s::%d-%d-%d' % (dtstr, em,
025 25        dn, choice(doms), dtint, shorter, longer)
026  <$nopage>
```

Running this code, we get the following output (your mileage will definitely vary) and store locally as the file `redata.txt`:

```
Thu Jul 22 19:21:19 2004::izsp@dicqdhytvhv.edu::1090549279-4-11
Sun Jul 13 22:42:11 2008::zqeu@dxaibjgkniy.com::1216014131-4-11
Sat May  5 16:36:23 1990::fclihw@alwdbzpsdg.edu::641950583-6-10
Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8
Thu Jun 26 19:08:59 2036::ugxfugt@jkhuqhs.net::2098145339-7-7
Tue Apr 10 01:04:45 2012::zkwaq@rpxwmtikse.com::1334045085-5-10
```

You may or may not be able to tell, but the output from this program is ripe for regular expression processing. Following our line-by-line explanation, we will implement several REs to operate on this data, as well as leave plenty for the end-of-chapter exercises.

## Line-by-line explanation

### Lines 1 – 6

In our example script, we require the use of multiple modules. But since we are utilizing only one or two functions from these modules, rather than importing the entire module, we choose in this case to import only specific attributes from these modules. Our decision to use **`from-import`** rather than **`import`** was based solely on this reasoning. The **`from-import`** lines succeed the UNIX start-up directive on line 1.

### Line 8

`doms` is simply a set of higher-level domain names from which we will randomly pick for each randomly-generated e-mail address.

### Lines 10–12

Each time `gendata.py` executes, between 5 and 10 lines of output are generated. (Our script uses the `random.randint()` function for all cases where we desire a random integer.) For each line, we choose a random integer from the entire possible range (0 to $2^{31}$ - 1 `[sys.maxint]`), then convert that integer to a date using `time.ctime()`.

### Lines 14–22

The login name for the fake e-mail address should be between 4 and 7 characters in length. To put it together, we randomly choose between 4 and 7 random lowercase letters, concatenating each letter to our string one-at-a-time. The functionality of the `random.choice()` function is given a sequence, return a random element of that sequence. In our case, the sequence is the set of all 26 lowercase letters of the alphabet, `string.lowercase`.

We decided that the main domain name for the fake e-mail address should be between 4 and 12 characters in length, but at least as long as the login name. Again, use random lowercase letters to put this name together letter-by-letter.

#### Line 24–25

The key component of our script puts together all of the random data into the output line. The date string comes first, followed by the delimiter. We then put together the random e-mail address by concatenating the login name, the "@" symbol, the domain name, and a randomly chosen high-level domain. After the final double-colon, we put together a random integer string using the original time chosen (for the date string), followed by the lengths of the login and domain names, all separated by a single hyphen.

## Matching a string

For the following exercises, create both permissive and restrictive versions of your REs. We recommend you test these REs in a short application which utilizes our sample `redata.txt` file above (or use your own generated data from running `gendata.py`). You will need to use it again when you do the exercises.

To test the RE before putting it into our little application, we will import the `re` module and assign one sample line from `redata.txt` to a string variable data. These statements are constant across both illustrated examples.

```
>>> import re
>>> data = Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-
8
```

In our first example, we will create a regular expression to extract (only) the days of the week from the timestamps from each line of the data file `redata.txt`. We will use the following RE:

```
"^Mon|^Tue|^Wed|^Thu|^Fri|^Sat|^Sun"
```

This example requires that the string start with ("^" RE operator) any of the seven strings listed. If we were to "translate" the above RE to English, it would read something like, “the string should start with "Mon," "Tue,"… , "Sat," or "Sun."

Alternatively, we can bypass all the carat operators with a single carat if we group the day strings like this:

```
"^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)"
```

The parentheses around the set of strings mean that one of these strings must be encountered for a match to succeed. This is a "friendlier" version of the original RE which we came up with which did not have the parentheses. Using our modified RE, we can take advantage of the fact that we can access the matched string as a subgroup:

```
>>> patt = '^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)'
>>> m = re.match(patt, data)
>>> m.group()                          # entire match
'Thu'
>>> m.group(1)                         # subgroup 1
'Thu'
>>> m.groups()                         # all subgroups
('Thu',)
```

This feature may not seem as revolutionary as we have made it out to be for this example, but it definitely advantageous in the next example or anywhere you provide extra data as part of the RE to help in the string matching process, even though those characters may not be part of the string you are interested in.

Both of the above REs are the most restrictive, specifically requiring a set number of strings. This may not work well in an internationalization environment where localized days and abbreviations are used. A looser RE would be:"`^\w{3}.`"This one requires only that a string begin with three consecutive alphanumeric characters. Again, to translate the RE into English, the carat indicates "begins with," the "\w" means any single alphanumeric character, and the "`{3}`" means that there should be 3 consecutive copies of the RE which the "`{3}`" embellishes. Again, if you want grouping, parentheses should be used, i.e., "`^(\w{3}):`"

```
>>> patt = '^(\w{3})'
>>> m = re.match(patt, data)
>>> if m != None: m.group()
…
'Thu'
>>> m.group(1)
'Thu'
```

Note that an RE of "`^(\w){3}`" is not correct. When the "`{3}`" was inside the parentheses, the match for 3 consecutive alphanumeric characters was made first, then represented as a group. But by moving the "`{3}`" outside, it is now equivalent to 3 consecutive single alphanumeric characters:

```
>>> patt = '^(\w){3}'
>>> m = re.match(patt, data)
>>> if m != None: m.group()
…
'Thu'
>>> m.group(1)
'u'
```

The reason why only the "u" shows up when accessing subgroup 1 is that subgroup 1 was being continually replaced by the next character. In other words, `m.group(1)` started out as "T," then changed to "h," then finally was replaced by "u." These are 3 individual (and overlapping) groups of a single alphanumeric character, as opposed to a single group consisting of 3 consecutive alphanumeric characters.

In our next (and final) example, we will create a regular expression to extract the numeric fields found at the end of each line of `redata.txt`.

## Search vs. Match

Before we create any REs, however, we realize that these integer data items are at the end of the data strings. This means that we have a choice of using either search or match. Initiating a search makes more sense because we know exactly what we are looking for (set of 3 integers), that what we seek is not at the beginning of the string, and that it does not make up the entire string. If we were to perform a match, we would have to create an RE to match the entire line and use subgroups to save the data we are interested in. To illustrate the differences, we will perform a search first, then do a match to show you that searching is more appropriate.

Since we are looking for 3 integers delimited by hyphens, we create our RE to indicate as such: "`\d+-\d+-\d+`". This regular expression means, “any number of digits (at least one, though) followed by a hyphen, then more digits, another hyphen, and finally, a final set of digits. We test our RE now using `search()`:

```
>>> patt = '\d+-\d+-\d+'
>>> re.search(patt, data).group()     # entire match
'1171590364-6-8'
```

A match attempt, however, would fail. Why? Because matches start at the beginning of the string, the numeric strings are at the rear. We would have to create another RE to match the entire string. We can be lazy though, by using ".+" to indicate just an arbitrary set of characters followed by what we are really interested in:

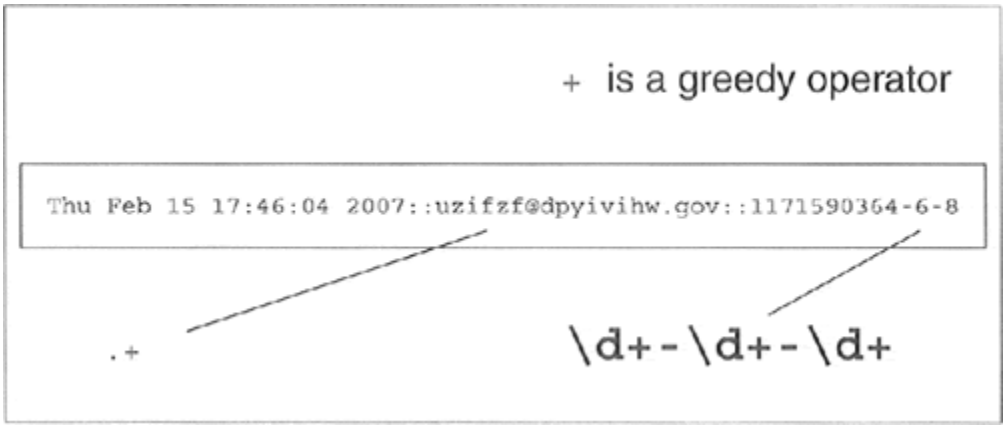
```
patt = '.+\d+-\d+-\d+'
>>> re.match(patt, data).group()    # entire match
'Thu Feb 15 17:46:04
2007::uzifzf@dpyivihw.gov::1171590364-6-8'
```

This works great, but we really want the number fields at the end, not the entire string, so we have to use parentheses to group what we want:

```
>>> patt = '.+(\d+-\d+-\d+)'
>>> re.match(patt, data).group(1)     # subgroup 1
'4-6-8'
```

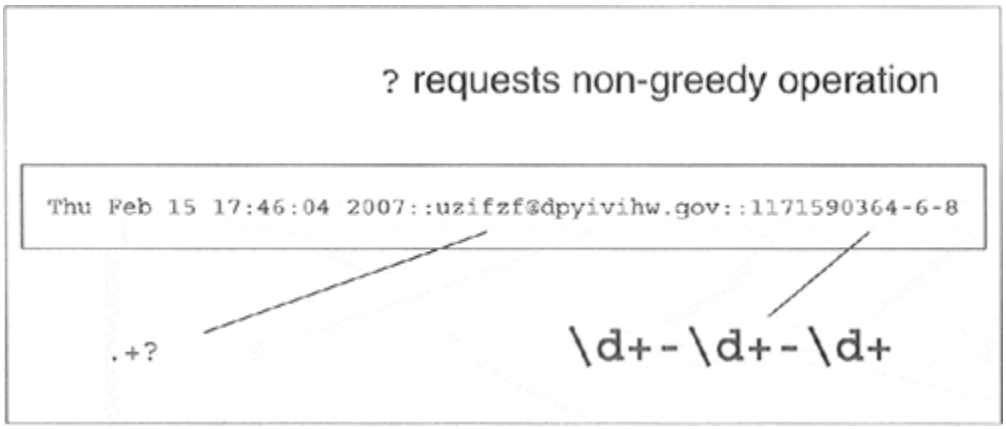
Eek! What happened? We should have extracted "1171590364-6-8," not just "4-6-8". Where is the rest of the first integer? The problem is that regular expressions are inherently "greedy." That means that with wildcard patterns, regular expressions are evaluated in left-to-right order and try to "grab" as many characters as possible which match the pattern. In our case above, the ".+" grabbed every single character from the beginning of the string, including most of the first integer field we wanted. The "\d+" needed only a single digit, so it got "4," while the ".+" matched everything from the beginning of the string up to that first digit: "Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::117159036," as indicated below in Figure 15-2.

**Figure 15.2.**



The solution is to use the "don't be greedy" operator, "?". It can be used after "*", "+", or "?". This directs the regular expression engine to match as few characters as possible. So if we place a "?" after the ".+", we obtain the desired result illustrated in Figure15-3.

**Figure 15.3. Solving the Greedy Problem:? Requests Non-Greediness**




```
>>> patt = '.+?(\d+-\d+-\d+)'
>>> re.match(patt, data).group(1)     # subgroup 1
'1171590364-6-8'
```

One final example. Let's say we want to pull out only the middle integer of the three-integer field. Here is how we would do it (using a search so we don't have to match the entire string): "-(\d+)-". Trying out this pattern, we get:

```
>>> patt = '-(\d+)-'
>>> m = re.search(patt, data)
>>> m.group()                  # entire match
'-6-'
>>> m.group(1)                 # subgroup 1
'6'
```

We barely touched upon the power of regular expressions, and in this limited space we have not been able to give them justice. However, we hope that we have given an informative introduction so that you can add this powerful tool to your programming skills. We suggest you refer to the documentation for more details on how to use REs with Python. For more complete immersion into the world of regular expressions, we recommend *Mastering Regular Expressions* by Jeffrey E. F. Friedl.

## Exercises

*Regular Expressions.* Create regular expressions in Exercises 15-1 to 15-12 to:

**1:**
Recognize the following strings: "bat," "bit," "but," "hat," "hit," or "hut."

**2:**
Match any pair of words separated by a single space, i.e., first and last names.

**3:**
Match any word and single letter separated by a comma and single space, as in last name, first initial.

**4:**
Match the set of all valid Python identifiers.

**5:**
Match a street address according to your local format (keep your RE general enough to match any number of street words, including the type designation). For example, American street addresses use the format: 1180 Bordeaux Drive. Make your RE general enough to support multi-word street names like: 3120 De la Cruz Boulevard.

**6:**
Match simple Web domain names that begin with "www." and end with a ".com" suffix, e.g., http://www.yahoo.com. EXTRA CREDIT if your RE also supports other high-level domain names: .edu, .net, etc., e.g., http://www.ucsc.edu

**7:**
Match the set of the string representations of all Python integers.

**8:**
Match the set of the string representations of all Python longs.

**9:**
Match the set of the string representations of all Python floats.

**10:**
Match the set of the string representations of all Python complex numbers.

**11:**
Match the set of all valid e-mail addresses (start with a loose RE, then try to tighten

it as much as you can, yet maintain correct functionality).

**12:**

Match the set of all valid Web site addresses (URLs) (start with a loose RE, then try to tighten it as much as you can, yet maintain correct functionality).

**13:**

*type()*. The `type()` built-in function returns a type object which is displayed as a Pythonic-looking string:

```
>>> type(0)
<type 'int'>
>>> type(.34)
<type 'loat'>
>>> type(dir)
<type 'builtin_function_or_method'>
```

Create an RE that would extract out the actual type name from the string. Your function should take a string like this " `<type 'int'>`" and return 'int'. (Ditto for all other types, i.e., 'float', 'builtin_function_or_method', etc.) Note: you are implement the value that is stored in the `__name__` attribute for classes and some built-in types.

**14:**

*Regular Expressions*. In Section 15.2, we gave you the RE pattern which matched the single- or double-digit string representations of the months January to September ("0?[1–9]"). Create the RE that represents the remaining three months in the standard calendar.

**15:**

*Regular Expressions*. Also in Section 15.2, we gave you the RE pattern which matched credit card (CC) numbers ("[0–9]{15,16}"). However, this pattern does not allow for hyphens separating blocks of numbers. Create the RE that allows hyphens, but only in the correct locations. For example, 15-digit CC numbers have a pattern of 4-6-5, indicating four digits-hyphen-six digits-hyphen-five digits, and 16-digit CC numbers have a 4-4-4-4 pattern. Remember to "balloon" the size of the entire string correctly. EXTRA CREDIT: there is a standard algorithm for determining whether a CC number is valid. Write some code to not only recognize a correctly formatted CC number, but also a valid one.

The next set of problems (15–16 through 15–27) deal specifically with the data that is generated *by `gendata.py`* Before approaching problems 15–17 and 15–18, you

may wish to do 15–16 and all the regular expressions first.

**16:**

Update the code for `gendata.py` so that the data is written directly to `redata.txt` rather than output to the screen.

**17:**

Determine how many times each day of the week shows up for any incarnation of `redata.txt`.(Alternatively, you can also count how many times each month of the year was chosen.)

**18:**

Ensure there is no data corruption in `redata.txt` by confirming that the first integer of the integer field matches the timestamp given at the front of each output line.

Create regular expressions to:

**19:**

Extract the complete timestamps from each line.

**20:**

Extract the complete e-mail address from each line.

**21:**

Extract only the months from the timestamps.

**22:**

Extract only the years from the timestamps.

**23:**

Extract only the time (HH:MM:SS) from the timestamps.

**24:**

Extract only the login and domain names (both the main domain name and the high-level domain together) from the e-mail address.

**25:**

Extract only the login and domain names (both the main domain name and the

high-level domain) from the e-mail address.

**26:**

Replace the e-mail address from each line of data with your e-mail address.

**27:**

Extract the months, days, and years from the timestamps and output them in "Mon Day, Year" format, iterating over each line only once.

For problems 15–28 and 15–29, recall the regular expression introduced in Section 15.2 which matched telephone numbers but allowed for an optional area code prefix: `\d{3}-\d{3}-\d{4}` Update this regular expression so that:

**28:**

Area codes (the first set of three-digits and the accompanying hyphen) are optional, i.e., your RE should match both 800-555-1212 as well as just 555-1212.

**29:**

Either parenthesized or hyphenated area codes are supported, not to mention optional; make your RE match 800-555-1212, 555-1212, and also (800) 555-1212.

# Chapter 16. Network Programming

In this section, we will take a brief look at network programming using sockets. We will first present some background information on network programming, how sockets apply to Python, then show you how to use some of Python's modules to build networked applications.

## Introduction

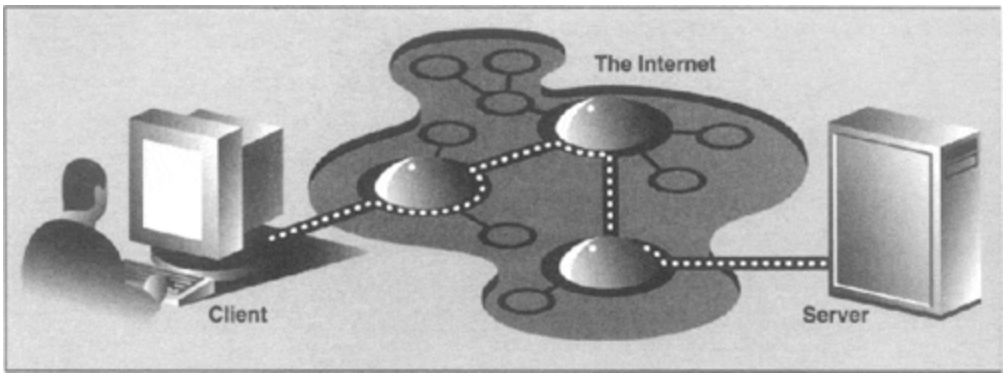
### What is Client-Server Architecture?

What is client-server architecture? It means different things to different people, depending on whom you ask as well as whether you are describing a software or a hardware system. In either case, the premise is simple: The *server,* a piece of hardware or software, is providing a "service" which is needed by one or more *clients,* users of the service. Its sole purpose of existence is to wait for (client) requests, service those clients, then wait for more requests.

Clients, on the other hand, contact a (predetermined) server for a particular request, send over any necessary data, and wait for the server to reply, either completing the request or indicating the cause of failure. While the server runs indefinitely processing requests, clients make a one-time request for service, receive that service, and thus conclude their transaction. A client may make additional requests at some later time, but these are considered separate transactions.

The most common notion of "client-server" today is illlustrated in Figure 16-1, a user or client computer is retrieving information from a server across the Internet. Although such a system is indeed an example of a client-server architecture, it isn't the only one. Furthermore, client-server architecture can be applied to computer hardware as well as software.

**Figure 16-1. Typical Conception of a Client-Server System on the Internet.**




**Hardware client-server architecture**

Print(er) servers are examples of hardware servers. They process incoming print jobs and send them to a printer (or some other printing device) attached to such a system. Such a computer is generally network-accessible and client machines would send print requests.

Another example of a hardware server is a file server. These are typically machines with large, generalized storage capacity which is remotely-accessible to clients. Client machines "mount" the disks from the server machine onto their local machine as if the disk itself were on the local machine. One of the most popular network operating systems which support file servers is Sun Microsystems' Network File System (NFS). If you are accessing a networked disk drive and cannot tell whether it is local or on the network, then the client-server system has done its job. The goal is for the user experience to be exactly the same as a local disk—the "abstraction" is normal disk access. It is up to the programmed "implementation" to make it behave in such a manner.

**Software client-server architecture**

Software servers also run on a piece of hardware but do not have dedicated peripheral devices as hardware servers do, i.e., printers, disk drives, etc. The primary services provided by software servers include program execution, data transfer retrieval, aggregation, update, or other type of programmed or data manipulation.

One of the more common software servers today is the Web server. A corporate machine is set up with Web pages and/or Web applications, then the Web server is started. The job of such a server is to accept client requests, send back Web pages to (Web) clients, i.e., browsers on users' computers, and wait for the next client request. These servers are started with the expectation of "running forever," although they do not achieve that goal, they go for as long as possible unless stopped by some external force, i.e., explicitly shut down or catastrophically due to hardware failure.

Database servers are another kind of software server. They take client requests for either storage or retrieval, perform that service, then wait for more business. They are also designed to run "forever."

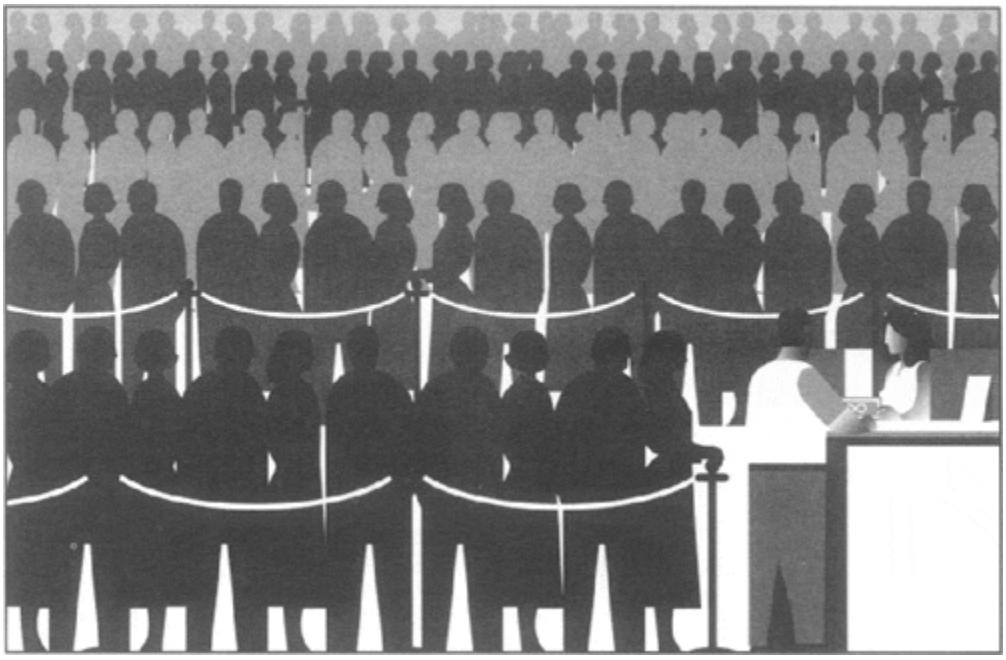
The last type of software server we will discuss are windows servers. These servers can almost be considered hardware servers. They run on a machine with an attached display, such as a monitor of some sort. Windows clients are actually programs which require a windowing environment with which to execute. These are generally considered graphical user interface (GUI) applications. If they are executed without a window server, i.e., in a text-based environment such as a DOS window or a Unix shell, they are unable to start. Once a windows server is accessible, then things are fine.

Such an environment becomes even more interesting when networking comes into play. The usual display for a windows client is the server on the local machine, but it is possible in some networked windowing environments, such as the X Windows system, to choose another machine's window server as a display. In such situations, you can be running a GUI program on one machine, but have it displayed at another!

### Bank tellers as servers?

One way to imagine how client-server architecture works is to create in your mind the image of a bank teller who neither eats, sleeps, nor rests, serving one customer after another in a line that never seems to end (see Figure 16-2). The line may be long or it may be empty on occasion, but at any given moment, a customer may show up. Of course, such a teller was fantasy years ago, but automated teller machines (ATMs) seem to come close to such a model now.

**Figure 16-2. The bank teller in this diagram works "forever" serving client requests. The teller runs in an infinite loop receiving requests, servicing them, and going back to serve or wait for another client. There may be a long line of clients, or there may be none at all, but in either case, a server's work is never done.**



The teller is, of course, the server that runs in an infinite loop. Each customer is a client with a need which requires servicing. Customers arrive and are serviced by the teller in a first-come-first-served manner. Once a transaction has been completed, the client goes away while the server either serves the next customer or sits and waits until one comes along.

Why is all this important? The reason is that this style of execution is how client-server architecture works in a general sense. Now that you have the basic idea, let us adapt it to network programming, which follows the software client-server architecture model.

### Client-Server Network Programming

Before any servicing can be accomplished, a server must perform some preliminary setup procedures to prepare for the work that lies ahead. A communication endpoint is created which allows a server to "listen" for requests. One can liken our server to a company receptionist or switchboard operator who answers calls on the main corporate line. Once the phone number and equipment are installed and the operator arrives, the service can begin.

This process is the same in the networked world—once a communication endpoint has been established, our listening server can now enter its infinite loop to wait for clients to connect and be serviced. Of course, we must not forget to put that phone number on company letterhead, in advertisements, or some sort of press release; otherwise, no one will ever call!

On a related note, potential clients must be made aware that this server exists to handle their needs—otherwise, the server will never get a single request. Imagine creating a brand new Web site. It may be the most super-duper, awesome, amazing, useful, and coolest Web site of all, but if the Web address or Uniform Resource Locator (URL) is never broadcast or advertised in any way, no one will ever know about it, and it will never see see the light of day. The same thing applies for the new telephone number of corporate headquarters. No calls will ever be received if the number is not made known to the public.

Now you have a good idea as to how the server works. You have gotten past the difficult part. The client side stuff is much more simple than on the server side. All the client has to do is to create its single communication endpoint, establish a connection to the server. The client can now make their request, which includes any necessary exchange of data. Once the request has been serviced and the client has received the result or some sort of acknowledgement, communication is terminated.

## Sockets: Communication Endpoints

### What Are Sockets?

Sockets are computer networking data structures which embody the concept of the "communication endpoint" described in the previous section. Networked applications

must create sockets before any type of communication can commence. They can be likened to telephone jacks, without which engaging in communication is impossible.

Sockets originated in the 1970s from the University of California, Berkeley version of UNIX, known as BSD UNIX. Therefore, you will sometimes hear these sockets referred to as "Berkeley sockets" or "BSD sockets." Sockets were originally created for same-host applications where they would enable one running program (a.k.a. a process) to communicate with another running program. This is known as *interprocess communication,* or "IPC" for short.

One interesting historical note: Sockets were invented before networking existed. Despite what you may have heard, sockets have not always been just for networked applications. These original sockets, still in use today, are called UNIX sockets and have a "family name" of AF_UNIX, which stands for "*address family:* UNIX." (Most popular platforms, including Python, use the term "address families" and "AF" abbreviation while other systems may refer to address families as "domains" or "protocol families" and use "PF" rather than "AF.") Because both processes run on the same machine, these sockets are file-based, meaning that their underlying infrastructure is supported by the file system. This makes sense because the file system is a shared constant between processes running on the same host.

When networking (utilizing the Internet Protocol [a.k.a. IP]) became a reality, researchers believed that interprocess communication should still be able to take place, but rather than restricting both applications to running on the same machine, why not enable a process on one machine to talk to a process on a different machine? These newer, networked sockets have their own family name, AF_INET, or "address family: Internet." There are other address families, all of which are either specialized, antiquated, seldom used, or remain unimplemented. Of all address families, AF_INET is now the most widely used. Python supports only the AF_UNIX and AF_INET families. Because of our focus on network programming, we will be using AF_INET for most of the remaining part of this chapter.

## Socket Addresses: Host-port Pairs

If a socket is like a telephone jack, a piece of infrastructure that enables communication, then a hostname and port number are like an area code and telephone number combination. Having the hardware and ability to communicate doesn't do any good unless you know whom and where to "dial." An Internet address is comprised of a hostname and port number pair, and such an address is required for networked communication. It goes without saying that there should also be someone listening at the other end; otherwise, you get the familiar, "DO-SO-DO" tones followed by "I'm sorry, that number is no longer in service. Please check the number and try your call again." You have probably seen one networking analogy during Web surfing, i.e., "Unable to contact server. Server is not responding or is unreachable."

Valid port numbers range from 0–65535, although those less than 1024 are reserved for the system. If you are using a Unix system, the list of reserved port numbers (along with

servers/protocols and socket types) is found in the `/etc/services` file. A list of well-known port numbers is accessible at this Web site:

http://www.isi.edu/in-notes/iana/assignments/port-numbers

## Connection-Oriented vs. Connectionless

### Connection-Oriented

Regardless of which address family you are using, there are two different styles of socket connections. The first type is connection-oriented. What this basically means is that a connection must be established before communication can occur, such as calling a friend using the telephone system. This type of communication is also referred to as a "virtual circuit" or "stream socket."

Connection-oriented communication offers sequenced, reliable, and unduplicated delivery of data, and without record boundaries. That basically means that each message may be broken up into multiple pieces, which are all guaranteed to arrive ("exactly-once" semantics means no loss or duplication of data) at their destination to be, put back together and in order, and delivered to the waiting application.

The primary protocol which implements such connection types is the Transmission Control Protocol (or better known by its acronym "TCP"). To create TCP sockets, one must use SOCK_STREAM as the type of socket one wants to create. The SOCK_STREAM name for a TCP socket is based on one of its denotations as stream socket. Because these sockets use the Internet Protocol to find hosts in the network, the entire system generally goes by the combined names of both protocols (TCP and IP) or "TCP/IP."

### Connectionless

In stark contrast to virtual circuits is the datagram type of socket, which is connectionless. This means that no connection is necessary before communication can begin. Here, there are no guarantees of sequencing, reliability or non-duplication in the process of data delivery. Datagrams do preserve record boundaries, however, meaning that entire messages are sent rather than being broken into pieces first, like connection-oriented protocols.

Message delivery using datagrams can be compared to the postal service. Letters and packages may not arrive in the order they were sent. In fact, they might not arrive at all! To add to the complication, in the land of networking, *duplication* of messages is even possible.

So with all this negativity, why use datagrams at all? (There must be *some* advantage over using stream sockets!) Because of the guarantees provided by connection-oriented

sockets, a good amount of overhead is required for their setup as well as in maintaining the virtual circuit connection. Datagrams do not have this overhead and thus are "less expensive," usually providing better performance and may be suitable for some types of applications.

The primary protocol which implements such connection types is the User Datagram Protocol (or better known by its acronym "UDP"). To create UDP sockets, one must use SOCK_DGRAM as the type of socket they want to create. The SOCK_DGRAM name for a UDP socket, as you can probably tell, comes from the word "datagram." Because these sockets also use the Internet Protocol to find hosts in the network, this system also has a more general name, going by the combined names of both of these protocols (UDP and IP), or "UDP/IP."

## Network Programming in Python

Now that you know all about client-server architecture, sockets, and networking, let us try to bring this concept to Python. The primary module we will be using in this section is the `socket` module. Found within this module is the `socket()` function, which is used to create socket objects. Sockets also have their own set of methods which enable socket-based network communication.

### `socket()` Module Function

To create a socket, you must use the `socket.socket()` function, which has the general syntax:

```
socket (socket_family, socket_type, protocol=0)
```

The *`socket_family`* is either AF_UNIX or AF_INET, as explained earlier, and the *`socket_type`* is either SOCK_STREAM or SOCK_DGRAM, also explained earlier. The *`protocol`* is usually left out, defaulting to 0.

So to create a TCP/IP socket, you call `socket.socket()` like this:

```
tcpSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Likewise, to create a UDP/IP socket you perform:

```
udpSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Since there are numerous `socket` module attributes, this is one of the exceptions where using "**`from`** *`module`* **`import *`**" is acceptable because of the number of module attributes. If we applied "**`from`** *`socket`* **`import *`**", we bring the socket attributes into our namespace, but our code is shortened considerably, i.e.,

```
tcpSock = socket(AF_INET, SOCK_STREAM)
```

Once we have a socket object, all further interaction will occur using that socket object's methods.

## Socket Object (Built-in) Methods

In Table 16.1, we present a list of the most common socket methods. In the next subsection, we will create both TCP and UDP clients and servers, all of which use these methods. Although we are focusing on Internet sockets, these methods have similar meanings when using Unix sockets.

**Table 16.1. Common Socket Object Methods**

| Method | *Description* |
|---|---|
| ***Server Socket Methods*** | |
| `s.bind()` | bind address (hostname, port number pair) to socket |
| `s.listen()` | set up and start TCP listener |
| `s.accept()` | passively accept TCP client connection, waiting until connection arrives (blocking) |
| ***Client Socket Methods*** | |
| `s.connect()` | actively initiate TCP server connection |
| ***General Socket Methods*** | |
| `s.recv()` | receive TCP message |
| `s.send()` | transmit TCP message |
| `s.recvfrom()` | receive UDP message |
| `s.sendto()` | transmit UDP message |
| `s.close()` | close socket |

## Creating a TCP Server

We will first present some general pseudocode involved with creating a generic TCP server, then describe in general what is going on. Keep in mind that this is only one way of designing your server. Once you become comfortable with server design, you will be able to modify the pseudocode to operate the way you want it to:

```
ss = socket()                   # create server socket
```

```
ss.bind()                       # bind socket to address
ss.listen()                     # listen for connections
inf_loop:                       # server infinite loop
    cs = ss.accept()            # accept client connection
    comm_loop:                  # communication loop
        cs.recv()/cs.send()     # dialog (receive/send)
    cs.close()                  # close client socket
ss.close()                      # close server socket
```

All sockets are created using the `socket.socket()` function. Servers need to "sit on a port" and wait for requests, so they all must "bind" to a local address. Because TCP is a connection-oriented communication system, some infrastructure must be set up before a TCP server can begin operation. In particular, TCP servers must "listen" for (incoming) connections. Once this setup process is complete, a server can start its infinite loop.

A simple (single-threaded) server will then sit on an `accept()` call waiting for a connection. By default, `accept()` is blocking, meaning that execution is suspended until a connection arrives. Sockets do support a non-blocking mode; refer to the documentation or operating systems textbooks for more details on why and how you would use non-blocking sockets.

Once a connection is accepted, a separate client socket is returned [by `accept()`] for the upcoming message interchange. Using the new client socket is similar to handing off a customer call to a service representative. When a client eventually does come in, the main switchboard operator takes the incoming call and patches it through, using another line to the right person to handle their needs.

This frees up the main line, i.e., the original server socket so that the operator can resume waiting for new calls (client requests) while the customer and the service representative he or she was connected to carry on their own conversation. Likewise, when an incoming request arrives, a new communication port is created to converse directly with that client while the main one is free to accept new client connections.

**NOTE**

We do not implement this in our examples, but it is also fairly common to hand a client request off to a thread or new process to complete the client processing. The `SocketServer` module, a high-level socket communication module written on top of `socket`, supports both threaded and spawned process handling of client requests. We refer the reader to the documentation to obtain more information about the `SocketServer` module as well as the exercises in *Chapter 17*, Multithreaded Programming.

Once the temporary socket is created, communication can commence, and both client and server proceed to engage in a dialog of sending and receiving using this new socket until the connection is terminated. This usually happens when one of the parties either closes its connection or sends an empty string to its partner.

In our code, after a client connection is closed, the server goes back to wait for another client connection. The final line of code, where we close the server socket, is never encountered since it is supposed to run in an infinite loop. We leave this code in our example as a reminder to the reader that calling the `close()` method is recommended when implementing an intelligent exit scheme for the server, for example, a handler which detects some external condition whereby the server should be shut down. In those cases, a `close()` method call is warranted.

In Example 16.1, we present `tsTserv.py`, a TCP server program which takes the data string sent from a client and returns it timestamped (format: "`[timestamp] data`") back to the client. ("tsTserv" stands for *t*ime*s*tamp *T*CP *serv*er. The other files are named in a similar manner.)

### Example 16.1. TCP Timestamp Server (`tsTserv.py`)

Creates a TCP server which accepts messages from clients and returns them with a timestamp prefix.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from socket import * <$nopage>
004 4  from time import time, ctime
005 5
006 6  HOST = ''
007 7  PORT = 21567
008 8  BUFSIZ = 1024
009 9  ADDR = (HOST, PORT)
010 10
011 11 tcpSerSock = socket(AF_INET, SOCK_STREAM)
012 12 tcpSerSock.bind(ADDR)
013 13 tcpSerSock.listen(5)
014 14
015 15 while 1:
016 16     print 'waiting for connection…'
017 17     tcpClisock, addr = tcpSerSock.accept()
018 18     print '…connected from:', addr
019 19
020 20     while 1:
021 21         data = tcpCliSock.recv(BUFSIZ)
022 22         if not data: break <$nopage>
023 23         tcpCliSock.send('[%s] %s' % \
024 24             ctime(time()), data)
025 25
026 26     tcpCliSock.close()
027 27 tcpSerSock.close()
028  <$nopage>
```

#### Line-by-line

#### Lines 1–4

After the Unix start-up line, we import `time.time()`, `time.ctime()`, and all the attributes from the `socket` module.

#### Lines 6–13

The `HOST` variable is blank, an indication to the `bind()` method that it can use any address that is available. We also choose an arbitrarily random port number which does not appear to be used or reserved by the system. For our application, we set the buffer size to 1K. You may vary this size based on your networking capability and application needs. The argument for the `listen()` method is simply a maximum number of incoming connection requests to accept before connections are turned away or refused.

The TCP server socket (`tcpSerSock`) is allocated on line 11, followed by the calls to bind the socket to the server's address and to start the TCP listener.

#### Lines 15–27

Once we are inside the server's infinite loop, we (passively) wait for a connection. When one comes in, we enter the dialog loop where we wait for the client to send its message. If the message is blank, that means that the client has quit, so we would break from the dialog loop, close the client connection, and go back to wait for another client. If we did get a message from the client, then we format and return the same data but prepended with the current timestamp. The final line (27) is never executed, but is there as a reminder to the reader that a `close()` call should be made if a handler is written to allow for a more graceful exit, as we discussed before.

### Creating a TCP Client

Creating a client is much simpler than a server. Similar to our description of the TCP server, we will present the pseudocode with explanations first, then show you the real thing.

```
cs = socket()               # create client socket
cs.connect()                # attempt server connection
comm_loop:                  # communication loop
    cs.send()/cs.recv()     # dialog (send/receive)
cs.close()                  # close client socket
```

As we noted before, all sockets are created using `socket.socket()`. Once a client has a socket, however, it can immediately make a connection to a server by using the socket's `connect()` method. When the connection has been established, then it can participate in

dialog with the server. Once the client has completed its transaction, it may close its socket, terminating the connection.

We present the code for `tsTclnt.py` in Example 16.2; it connects to the server and prompts the user for line-after-line of data. The server returns this data timestamped, which is presented to the user by the client code.

### Example 16.2. TCP Timestamp Client (`tsTclnt.py`)

Creates a TCP client which prompts the user for messages to send to the server, gets them back with a timestamp prefix, and displays the results to the user.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from socket import * <$nopage>
004 4
005 5  HOST = 'localhost'
006 6  PORT = 21567
007 7  BUFSIZ = 1024
008 8  ADDR = (HOST, PORT)
009 9
010 10 tcpCliSock = socket(AF_INET, SOCK_STREAM)
011 11 tcpCliSock.connect(ADDR)
012 12
013 13 while 1:
014 14     data = raw_input('> ')
015 15     if not data: break <$nopage>
016 16     tcpCliSock.send(data)
017 17     data = tcpCliSock.recv(1024)
018 18     if not data: break <$nopage>
019 19     print data
020 20
021 21 tcpCliSock.close()
022  <$nopage>
```

#### Line-by-line

#### Lines 1–3

After the Unix start-up line, we import all the attributes from the `socket` module.

#### Lines 5–11

The `HOST` and `PORT` variables refer to the server's hostname and port number. Since we are running our test (in this case) on the same machine, `HOST` contains the local hostname (change it accordingly if you are running your server on a different host). The port number `PORT` should be exactly the same as what you set for your server (otherwise there won't be much communication[!]). We also choose the same buffer size, 1K.

The TCP client socket (`tcpCliSock`) is allocated on line 10, followed by the call to connect to the server.

#### Lines 13–21

The client also has an infinite loop, but it is not meant to run forever like the server's loop. The client loop will exit on either of two conditions: If the user enters no input (line 15), or if the server somehow quit and our call to the `recv()` method fails (line 18). Otherwise, in a normal situation, the user enters in some string data, which is sent to the server for processing. The newly-timestamped input string is then received and displayed to the screen.

## Executing Our TCP Client-server Application

Now let's run the server and client programs to see how they work. Should we run the server first or the client first? Naturally, if we ran the client first, no connection would be possible because there is no server waiting to accept the request. The server is considered a "passive" partner because it has to establish itself first and passively wait for a connection. A client on the other hand is an "active" partner because it actively initiates a connection. In other words:

Start the server first (before any clients try to connect).

In our example running of the client and server, we use the same machine, but there is nothing to stop us from using another host for the server. If this is the case, then just change the hostname. (It is rather exciting when you get your first networked application running the server and client from different machines!)

We now present the corresponding (input and) output from the client program, which exits with a simple RETURN (or Enter key) keystroke with no data entered:

```
% tsTclnt.py
> hi
[Sat Jun 17 17:27:21 2000] hi
> spanish inquisition
[Sat Jun 17 17:27:37 2000] spanish inquisition
>
%
```

The server's output is mainly diagnostic:

```
% tsTserv.py
waiting for connection…
…connected from: ('127.0.0.1', 1040)
waiting for connection…
```

The "…connected from…" message was received when our client made its connection. The server went back to wait for new clients while we continued receiving "service." When we exited from the server, we had to break out of it, resulting in an exception. The best way to avoid such an error is to create a more graceful exit, as we have been discussing.

**NOTE**

One way to create this "friendly" exit is to put the server's **while** loop inside the **except** clause of a **try-except** statement and monitor for `EOFError` or `KeyboardInterrupt` exceptions. Then in the **except** clause, you can make a call to close the server's socket.

The interesting thing about this simple networked application is that we are not only showing how our data takes a round trip from the client to the server and back to the client, but we also use the server as a sort of "time server," because the timestamp we receive is purely from the server.

## Creating a UDP Server

UDP servers do not require as much setup as TCP servers because they are not connection-oriented. There is virtually no work that needs to be done other than just waiting for incoming connections.

```
ss = socket()                           # create server socket
ss.bind()                               # bind server socket
inf_loop:                               # server infinite loop
    cs = ss.recvfrom()/ss.sendto()      # dialog (receive/send)
ss.close()                              # close server socket
```

As you can see from the pseudocode, there is nothing extra other than the usual create-the-socket and bind it to the local address (host/port pair). The infinite loop consists of receiving a message from a client, returning a timestamped one, then going back to wait for another message. Again, the `close()` method will not be reached due to the infinite loop, but serves as a reminder that it should be part of the graceful or intelligent exit scheme we've been mentioning.

One other significant different between UDP and TCP servers is that because datagram sockets are connectionless, there is no "handing off" of a client connection to a separate socket for succeeding communication. These servers just accept messages and perhaps reply.

You will find the code to `tsUserv.py` in Example 16.3, a UDP version of the TCP server seen earlier. It accepts a client message and returns it to the client timestamped.

**Example 16.3. UDP Timestamp Server (`tsUserv.py`)**

Creates a UDP server which accepts messages from clients and returns them with a timestamp prefix.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from socket import * <$nopage>
004 4  from time import time, ctime
005 5
006 6  HOST = ''
007 7  PORT = 21567
008 8  BUFSIZ = 1024
009 9  ADDR = (HOST, PORT)
010 10
011 11 udpSerSock = socket(AF_INET, SOCK_DGRAM)
012 12 udpSerSock.bind(ADDR)
013 13
014 14 while 1:
015 15     print 'waiting for message…'
016 16     data, addr = udpSerSock.recvfrom(BUFSIZ)
017 17     udpSerSock.sendto('[%s] %s' % \
018 18         (ctime(time()), data), addr)
019 19     print '…received from, returned to:', addr
020 20
021 21 udpSerSock.close()
022  <$nopage>
```

**Line-by-line**

**Lines 1–4**

After the Unix start-up line, we import `time.time()`, `time.ctime()`, and all the attributes from the `socket` module, just like the TCP server setup.

**Lines 6–12**

The `HOST` and `PORT` variables are the same as before, and for all the same reasons. The call `socket()` differs only in that we are now requesting a datagram/UDP socket type, but `bind()` is invoked in the same way as in the TCP server version. Again, because UDP is connectionless, no call to "listen() for incoming connections" is made here.

**Lines 14–21**

Once we are inside the server's infinite loop, we (passively) wait for a connection. When one comes in, we process it (by adding a timestamp to it), then send it right back and go

back to wait for another message. The socket `close()` method is there for show only, as indicated before.

## Creating a UDP Client

Of the four highlighted here in this section, the UDP client is the shortest bit of code which we will look at. The pseudocode looks like this:

```
cs = socket()                    # create client socket
comm_loop:                       # communication loop
    cs.sendto()/cs.recvfrom()    # dialog (send/receive)
cs.close()                       # close client socket
```

Once a socket object is created, we enter the dialog loop of exchanging messages with the server. When communication is complete, the socket is closed.

The real client code, `tsUclnt.py`, is presented in Example 16.4.

### Example 16.4. UDP Timestamp Client (`tsUclnt.py`)

Creates a UDP client which prompts the user for messages to send to the server, gets them back with a timestamp prefix, and displays them back to the user.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from socket import * <$nopage>
004 4
005 5  HOST = 'localhost'
006 6  PORT = 21567
007 7  BUFSIZ = 1024
008 8  ADDR = (HOST, PORT)
009 9
010 10 udpCliSock = socket(AF_INET, SOCK_DGRAM)
011 11
012 12 while 1:
013 13     data = raw_input('> ')
014 14     if not data: break <$nopage>
015 15     udpCliSock.sendto(data, ADDR)
016 16     data, ADDR = udpCliSock.recvfrom(BUFSIZ)
017 17     if not data: break <$nopage>
018 18     print data
019 19
020 20 udpCliSock.close()
021  <$nopage>
```

#### Line-by-line

#### Lines 1–3

After the Unix start-up line, we import all the attributes from the `socket` module, again, just like in the TCP version of the client.

#### Lines 5–10

Because we are running the server on our local machine again, we use "localhost" and the same port number on the client side, not to mention the same 1K buffer. We allocate our socket object in the same way as the UDP server.

#### Lines 12–20

Our UDP client loop works almost exactly in the same manner as the TCP client. The only difference is that we do not have to establish a connection to the UDP server first, we simply send a message to it and await the reply. After the timestamped string is returned, we display it to the screen and go back for more. When the input is complete, we break out of the loop and close the socket.

## Executing Our UDP Client-Server Application

The UDP client behaves the same as the TCP client:

```
% tsUclnt.py
> hi
[Sat Jun 17 19:55:36 2000] hi
> spam! spam! spam!
[Sat Jun 17 19:55:40 2000] spam! spam! spam!
>
%
```

Likewise for the server:

```
% tsUserv.py
waiting for message…
… received from and returned to: ('27.0.0.1' 1025)
waiting for message…
```

In fact, we output the client's information because we can be receiving messages from multiple clients and sending replies, and such output helps in telling us where messages came from. With the TCP server, we know where messages come from because each client makes a connection. Note how the messages says, "waiting for message" as opposed to "waiting for connection."

## Other `socket` Module Functions

In addition to the `socket.socket()` function which creates a socket object, the `socket` module features a whole host of other ancillary functions to aid you in your networked applications, as seen below in Table 16.2.

**Table 16.2. Other `socket` Module Functions**

| *Function Name* | *Description* |
|---|---|
| `fromfd()` | create a socket object from an open file descriptor |
| `gethostname()` | return the current hostname |
| `gethostbyname()` | map a hostname to its IP number |
| `gethostbyaddr()` | map an IP number or hostname to DNS info |
| `getservbyname()` | map a service name and a protocol name to a port number |
| `getprotobyname()` | map a protocol name (e.g. 'tcp') to a number |
| `ntohl()/ntohs()` | converts integers from network to host byte order |
| `htonl()/htons()` | converts integers from host to network byte order |
| `inet_aton()` | convert IP address octet string to 32-bit packed format |
| `inet_ntoa()` | convert 32-bit packed format to IP address string |
| `ssl()` | Secure Socket Layer support (must be configured); new in 1.6 |

For more information, we refer you to the `socket` Module documentation in the Python Library Reference.

## Related Modules

Table 16.3 lists some of the other Python modules which are related to network and socket programming. The `select` module is usually used in conjunction with the `socket` module when developing lower-level socket applications. It provides the `select()` function which manages sets of socket objects. One of the most useful things it does is to take a set of sockets and listen for active connections on them. The `select()` function will block until at least one socket is ready for communication, and when that happens, it provides you with a set of which ones are ready for reading. (It can also determine which are ready for writing, although that is not as common as the former operation.)

**Table 16.3. Network/Socket Programming Related Modules**

| *Module* | *Description* |
|---|---|
| `asyncore` | provides infrastructure to create networked applications which process clients asynchronously |
| `select` | manages multiple socket connections in a single-threaded network server application |
| `SocketServer` | high-level module which provides server classes for networked applications, complete with forking or threading varieties |

The `asyncore` and `SocketServer` modules both provide higher-level functionality as far as create servers are concerned. Written on top of the `socket` and/or `select` modules, they enable more rapid development of client-server systems because all the lower-level code is handled for you. All you have to do is to create or subclass the appropriate base

classes, and you are on your way. As we mentioned earlier, `SocketServer` even provides the capability of integrating threading or new processes into the server for more parallelized processing of client requests.

The topics which we have covered in this chapter deal with network programming with sockets in Python and how to create custom applications using lower-level protocol suites such as TCP/IP and UDP/IP. If you want to develop higher-level Web and Internet applications, we strongly encourage you to head to Chapter 19.

## Exercises

**1:**

*Sockets.* What is the difference between connection-oriented versus connectionless?

**2:**

*Sockets.* What is the difference between TCP and UDP?

**3:**

*Sockets.* Between TCP and UDP, which type of servers accept connections and hands them off to separate sockets for client communication?

**4:**

*Clients.* Update the TCP (`tsTclnt.py`) and UDP (`tsUclnt.py`) clients so that the server name is not hard-coded into the application. Allow the user to specify a hostname and port number, and only use the default values if either or both parameters are missing.

**5:**

*Internetworking and Sockets.* Implement Guido's sample TCP client/server programs found in Section 7.2.2 of the Python Library Reference and get them to work. Set up the server, then the client. An online version of the source is also available here:

http://www.python.org/doc/current/lib/Socket_Example.html

You decide the server is too boring. Update the server so that it can do much more, recognizing the following commands:

| | |
|---|---|
| `date` | server will return its current date/timestamp, i.e., `time.ctime(time.time())` |

| | |
|---|---|
| `os` | get OS info (`os.name`) |
| `ls` | give a listing of the current directory (HINTS: `os.listdir()` lists a directory, `os.curdir` is the current directory) EXTRA CREDIT: accept "`ls dir`" and return `dir's` file listing |

You do not need a network to do this assignment—your machine can talk to itself. Note: After the server exits, the binding must be cleared before you can run it again. You may experience "port already bound" errors. The operating system usually clears the binding within 5 minutes, so be patient!

**6:**

*Daytime Service.* Use the `socket.getservbyname()` to determine the port number for the "daytime" service under the UDP protocol. Check the documentation for `getservbyname()` to get the exact usage syntax (i.e., `socket.getservbyname.__doc__`). Now write an application which sends a dummy message over and wait for the reply. Once you have received a reply from the server, display it to the screen.

**7:**

*Half Duplex Chat.* Create a simple, half-duplex chat program. By "half-duplex," we mean that when a connection is made and the service starts, only one person can type. The other participant must wait to get a message before he or she is prompted to enter a message. Once a message is sent, then the sender must wait for a reply before being allowed to send another message. One participant will be on the server side, while the other will be on the client side.

**8:**

*Full Duplex Chat.* Update your solution to the previous problem so that your chat service is now full-duplex, meaning that both parties can send and receive independently of each other.

**9:**

*Multi-User Full Duplex Chat.* Further update your solution so that your chat service is multi-user.

**10:**

*Multi-User Multi-Room Full Duplex Chat.* Now make your chat service multi-user and multi-room.

**11:**

*Web Client.* Write a TCP client which connects to port 80 of your favorite Web site (remove the "http://" and any trailing info; use only the hostname). Once a

connection has been established, send the HTTP command string "`GET /\n`" and write all the data that the server returns to a file. (The `GET` command retrieves a Web page, the "`/`" file indicates the file to get, and the "`\n`" sends the command to the server.) Examine the contents of the retrieved file. What is it? How can you check to make sure the data you received is correct? (Note: You may have to give one or two NEWLINEs of the command string. One usually works.)

**12:**

*Sleep Server.* Create a "sleep" server. A client will request to be "put to sleep" for a number of seconds. The server will issue the command on behalf of the client, then return a message to the client indicating success. The client should have slept or have been idle for the exact time requested. This is a simple implementation of a "remote procedure call" where a client's request invokes commands on another machine across the network.

**13:**

*Name Server.* Design and implement a name server. Such a server is responsible for maintaining a database of hostname-port number pairs; perhaps along with the string description of the service that the corresponding servers provide. Take one or more existing servers and have them "register" their service with your name server. (Note that these servers are, in this case, clients of the name server.)

Every client that starts up has no idea where the server is that it is looking for. Also as clients of the name server, these clients should send a request to the name server indicating what type of service they are seeking. The name server, in reply, returns a hostname-port number pair to this client, which then connects to the appropriate server to process its request.

EXTRA CREDIT: (1) add caching to your name server for popular requests, (2) add logging capability to your name server, keeping track of which servers have registered and which services clients are requesting; (3) your name server should periodically "ping" the registered hosts at their respective port numbers to ensure that the service is indeed up. Repeated failures will cause a server to be delisted from the list of services.

You may implement real services for the servers which register for your name service, or just use dummy servers (which merely acknowledge a request)

# Chapter 17. Multithreaded Programming

In this section, we will explore the different ways you can achieve more parallelism in your code by using the multithreaded (MT) programming features found in Python. We will begin by differentiating between processes and threads in the first few of sections of this chapter. We will then introduce the notion of multithreaded programming. (Those of you already familiar with MT programming can skip directly to Section 17.3.5.) The last section of this chapter lays out some examples of how to use the `threading` and `Queue` modules to accomplish MT programming with Python.

## Introduction/Motivation

Before the advent of multithreaded (MT) programming, running of computer programs consisted of a single sequence of steps which were executed in synchronous order by the host's central processing unit (CPU). This style of execution was the norm whether the task itself required the sequential ordering of steps or if the entire program was actually an aggregation of multiple subtasks. What if these subtasks were independent, having no causal relationship (meaning that results of subtasks do not affect other subtask outcomes)? Is it not logical, then, to want to run these independent tasks all at the same time? Such parallel processing could significantly improve the performance of the overall task. This is what MT programming is all about.

MT programming is ideal for programming tasks that are asynchronous in nature, require multiple concurrent activities, and where the processing of each activity may be *nondeterministic,* i.e., random and unpredictable. Such programming tasks can be organized or partitioned into multiple streams of execution where each has a specific task to accomplish. Depending on the application, these subtasks may calculate intermediate results that could be merged into a final piece of output.

While CPU-bound tasks may be fairly straightforward to divide into subtasks and executed sequentially or in a multithreaded manner, the task of managing a single-threaded process with multiple external sources of input is not as trivial. To achieve such a programming task without multithreading, a sequential program must use one or more timers and implement a multiplexing scheme.

A sequential program will need to sample each I/O (input/output) terminal channel to check for user input; however, it is important that the program does not block when reading the I/O terminal channel because the arrival of user input is nondeterministic, and blocking would prevent processing of other I/O channels. The sequential program must use non-blocked I/O or blocked I/O with a timer (so that blocking is only temporary).

Because the sequential program is a single thread of execution, it must juggle the multiple tasks that it needs to perform, making sure that it does not spend too much time on any one task, and it must ensure that user response time is appropriately distributed. The use of a sequential program for this type of programming task often results in a complicated flow of control program that is difficult to understand and maintain.

Using an MT program with a shared data structure such as a `Queue` (a multithreaded queue data structure discussed later in this chapter), this programming task can be organized with a few threads that have specific functions to perform:

**UserRequestThread:** responsible for reading client input, perhaps from an I/O channel. A number of threads would be created by the program, one for each current client, with requests being entered into the queue.

**RequestProcessor:** a thread that is responsible for retrieving requests from the queue and processing them, providing output for yet a third thread.

**ReplyThread:** responsible for taking output destined for the user and either sending it back, if in a networked application, or writing data to the local file system or database.

Organizing this programming task with multiple threads reduces the complexity of the program and enables an implementation that is clean, efficient, and well-organized. The logic in each thread is typically less complex because it has a specific job to do. For example, the **UserRequestThread** simply reads input from a user and places the data into a queue for further processing by another thread, etc. Each thread has its own job to do; and you merely have to design each type of thread to do one thing and do it well. Use of threads for specific tasks is not unlike Henry Ford's assembly line model for manufacturing automobiles.

## Threads and Processes

### What Are Processes?

Computer *programs* are merely executables, binary (or otherwise), which reside on disk. They do not take on a life of their own until loaded into memory and invoked by the operating system. A *process* (sometimes called a *heavyweight process*) is a program in execution. Each process has its own address space, memory, a data stack, and other auxiliary data to keep track of execution. The operating system manages the execution of all processes on the system, dividing the time fairly between all processes. Processes can also *fork* or *spawn* new processes to perform other tasks, but each new process has its own memory, data stack, etc., and cannot generally share information unless interprocess communication (IPC) is employed.

### What Are Threads?

*Threads* (sometimes called *lightweight processes*) are similar to processes except that they all execute within the same process, thus all share the same context. They can be

thought of as "mini-processes" running in parallel within a main process or "main thread."

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running. It can be pre-empted (interrupted) and temporarily put on hold (also known as *sleeping*) while other threads are running—this is called *yielding.*

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes. Threads are generally executed in a concurrent fashion, and it is this parallelism and data sharing that enable the coordination of multiple tasks. Naturally, it is impossible to run truly in a concurrent manner in a single CPU system, so threads are scheduled in such a way that they run for a little bit, then yield to other threads (going to the proverbial "back-of-the-line" to await getting more CPU time again). Throughout the execution of the entire process, each thread performs its own, separate tasks, and communicates the results with other threads as necessary.

Of course, such sharing is not without its dangers. If two or more threads access the same piece of data, inconsistent results may arise because of the ordering of data access. This is commonly known as a *race condition.* Fortunately, most thread libraries come with some sort of synchronization primitives which allow the thread manager to control execution and access.

Another caveat is that threads may not be given equal and fair execution time. This is because some functions block until they have completed. If not written specifically to take threads into account, this skews the amount of CPU time in favor of such greedy functions.

## Threads and Python

### Global Interpreter Lock

Execution by Python code is controlled by the *Python Virtual Machine* (a.k.a. the interpreter main loop), and Python was designed in such a way that only one thread of control may be executing in this main loop, similar to how multiple processes in a system share a single CPU. Many programs may be in memory, but only *one* is live on the CPU at any given moment. Likewise, although multiple threads may be "running" within the Python interpreter, only one thread is being executed by the interpreter at any given time.

Access to the Python Virtual Machine is controlled by a *global interpreter lock* (GIL). This lock is what ensures that exactly one thread is running. The Python Virtual Machine executes in the following manner in an MT environment:

Set the GIL,

Switch in a thread to run,

Execute for a specified number of bytecode instructions,

Put the thread back to sleep (switch out thread),

Unlock the GIL, and,

Do it all over again (rinse, lather, repeat).

When a call is made to external code, i.e., any C/C++ extension built-in function, the GIL will be locked until it has completed (since there are no Python bytecodes to count as the interval). Extension programmers do have the ability to unlock the GIL however, so you being the Python developer shouldn't have to worry about your Python code locking up in those situations.

As an example, for any Python I/O-oriented routines (which invoke built-in operating system C code), the GIL is released before the I/O call is made, allowing other threads to run while the I/O is being performed. Code which *doesn't* have much I/O will tend to keep the processor (and GIL) to the full interval a thread is allowed before it yields. In other words, I/O-bound Python programs stand a much better chance of being able to take advantage of a multithreaded environment than CPU-bound code.

Those of you interested in the source code, the interpreter main loop, and the GIL can take a look at `eval_code2()` routine in the `Python/ceval.c` file, which is the Python Virtual Machine.

## Exiting Threads

When a thread completes execution of the function they were created for, they exit. Threads may also quit by calling an exit function such as `thread.exit()`, or any of the standard ways of exiting a Python process, i.e., `sys.exit()` or raising the `SystemExit` exception.

There are a variety of ways of managing thread termination. In most systems, when the main thread exits, all other threads die without cleanup, but for some systems, they live on. Check your operating system threaded programming documentation regarding their behavior in such occasions.

Main threads should always be good managers, though, and perform the task of knowing what needs to be executed by individual threads, what data or arguments each of the spawned threads requires, when they complete execution, and what results they provide. In so doing, those main threads can collate the individual results into a final conclusion.

## Accessing Threads From Python

Python supports multithreaded programming, depending on the operating system that it is running on. It is supported on most versions of Unix, including Solaris and Linux, and

Windows. Threads are not currently available on the Macintosh platform. Python uses POSIX-compliant threads, or "pthreads," as they commonly known.

By default, threads are not enabled when building Python from source, but are available for Windows platforms automatically from the installer. To tell whether threads are installed, simply attempt to import the `thread` module from the interactive interpreter. No errors occur when threads are available:

```
>>> import thread
>>>
```

If your Python interpreter was not compiled with threads enabled, the module import fails:

```
>>> import thread
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named thread
```

In such cases, you may have to recompile your Python interpreter to get access to threads. This usually involves invoking the configure script with the "`--with-thread`" option. Check the `README` file for your distribution for specific instructions on how to compile Python with threads for your system.

Due to the brevity of this chapter, we will give you only a quick introduction to threads and MT programming in Python. We refer you to the official documentation to get the full coverage of all the aspects of the threading support which Python has to offer. Also, we recommended accessing any general operating system textbook for more details on processes, interprocess communication, multi-threaded programming, and thread/process synchronization. (Some of these texts are listed in the appendix.)

## Life Without Threads

For our first set of examples, we are going to use the `time.sleep()` function to show how threads work. `time.sleep()` takes a floating point argument and "sleeps" for the given number of seconds, meaning that execution is temporarily halted for the amount of time specified.

Let us create two "time loops," one which sleeps for 4 seconds and one that sleeps for 2 seconds, `loop0()` and `loop1()`, respectively. (We use the names "loop0" and "loop1" as a hint that we will eventually have a sequence of loops.) If we were to execute `loop0()` and `loop1()` sequentially in a one-process or single-threaded program, as `onethr.py` does in Example 17.1, the total execution time would be at least 6 seconds. There may or

may not be a 1-second gap between the starting of `loop0()` and `loop1()`, and other execution overhead which may cause the overall time to be bumped to 7 seconds.

**Example 17.1. Loops Executed by a Single Thread (`onethr.py`)**

Executes two loops consecutively in a single-threaded program. One loop must complete before the other can begin. The total elapsed time is the sum of times taken by each loop.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from time import sleep, time, ctime
004 4
005 5  def loop0():
006 6      print 'start loop 0 at:', ctime(time())
007 7      sleep(4)
008 8      print 'loop 0 done at:', ctime(time())
009 9
010 10 def loop1():
011 11     print 'start loop 1 at:', ctime(time())
012 12     sleep(2)
013 13     print 'loop 1 done at:', ctime(time())
014 14
015 15 def main():
016 16     print 'starting…'
017 17     loop0()
018 18     loop1()
019 19     print 'all DONE at:', ctime(time())
020 20
021 21 if __name__ == '__main__':
022 22     main()
023  <$nopage>
```

We can verify this by executing `onethr.py`, which gives the following output:

```
% onethr.py
starting…
start loop 0 at: Sun Aug 13 05:03:34 2000
loop 0 done at: Sun Aug 13 05:03:38 2000
start loop 1 at: Sun Aug 13 05:03:38 2000
loop 1 done at: Sun Aug 13 05:03:40 2000
all DONE at: Sun Aug 13 05:03:40 2000
```

Now, pretend that rather than sleeping, `loop0()` and `loop1()` were separate functions that performed individual and independent computations, all working to arrive at a common solution. Wouldn't it be useful to have them run in parallel to cut down on the overall running time? That is the premise behind MT that we will now introduce you to.

## Python Threading Modules

Python provides several modules to support MT programming, including the `thread`, `threading`, and `Queue` modules. The `thread` and `threading` modules allow the programmer to create and manage threads. The `thread` module provides the basic thread and locking support, while `threading` provides high-level full-featured thread management. The `Queue` module allows the user to create a queue data structure which can be shared across multiple threads. We will take a look at these modules individually, present a good number of examples, and a couple of intermediate-sized applications.

**NOTE**

We recommend avoiding the `thread` module for many reasons. The first is that the high-level `threading` module is more contemporary, not to mention the fact that thread support in the `threading` module is much improved and the use of attributes of the `thread` module may conflict with using the `threading` module. Another reason is that the lower-level `thread` module has a few synchronization primitives (actually only one) while `threading` has many.

However, in the interest of learning Python and threading in general, we do present some code which uses the `thread` module. These pieces of code should be used for learning purposes only and will give you a much better insight as to why you would want to avoid using the `thread` module. These examples also show how our applications and thread programming improve as we migrate to using more appropriate tools such as those available in the `threading` and `Queue` modules.

Use of the `thread` module is recommended only for experts desiring lower-level thread access. Those of you new to threads should look at the code samples to see how we can overlay threads onto our time loop application and to gain a better understanding as to how these first examples evolve to the main code samples of this chapter. Your first multithreaded application should utilize `threading` and perhaps other high-level thread modules, if applicable.

## `thread` Module

Let's take a look at what the `thread` module has to offer. In addition to being able to spawn threads, the `thread`module also provides a basic synchronization data structure called a *lock object* (a.k.a. primitive lock, simple lock, mutual exclusion lock, mutex, binary semaphore). As we mentioned earlier, such synchronization primitives go hand-in-hand with thread management.

Listed in Table 17.1 are a list of the more commonly-used thread functions and `LockType` lock object methods:

**Table 17.1. `thread` Module and Lock Objects**

| *Function/Method* | *Description* |
|---|---|
| **`thread` *Module Functions*** | |

| start_new_thread(*function, args, kwargs*=None) | spawns a new thread and execute *function* with the given *args* and optional *kwargs* |
|---|---|
| allocate_lock() | allocates LockType lock object |
| exit() | instructs a thread to exit |
| **LockType *Lock Object Methods*** | |
| acquire(*wait*=None) | attempts to acquire lock object |
| locked() | returns 1 if lock acquired, 0 otherwise |
| release() | releases lock |

The key function of the thread module is start_new_thread(). Its syntax is exactly that of the apply() built-in function, taking a function along with arguments and optional keyword arguments. The difference is that instead of the main thread executing the function, a new thread is spawned to invoke the function.

Let's take our onethr.py example and integrate threading into it. By slightly changing the call to the loop*() functions, we now present mtsleep1.py in Example 17.2.

### Example 17.2. Using the thread Module (mtsleep1.py)

The same loops from onethr.py are executed, but this time using the simple multithreaded mechanism provided by the thread module. The two loops are executed concurrently (with the shorter one finishing first, obviously), and the total elapsed time is only as long as the slowest thread rather than the total time for each separately.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import thread
004 4  from time import sleep, time, ctime
005 5
006 6  def loop0():
007 7      print 'start loop 0 at:', ctime(time())
008 8      sleep(4)
009 9      print 'loop 0 done at:', ctime(time())
010 10
011 11 def loop1():
012 12     print 'start loop 1 at:', ctime(time())
013 13     sleep(2)
014 14     print 'loop 1 done at:', ctime(time())
015 15
016 16 def main():
017 17     print 'starting threads…'
018 18     thread.start_new_thread(loop0, ())
019 19     thread.start_new_thread(loop1, ())
020 20     sleep(6)
021 21     print 'all DONE at:', ctime(time))
022 22
023 23 if __name__ == '__main__':
024 24     main()
025  <$nopage>
```

`start_new_thread()` requires the first two arguments, so that's the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4, the length of time of our longest loop, plus any overhead.

```
% mtsleep1.py
starting threads…
start loop 0 at: Sun Aug 13 05:04:50 2000
start loop 1 at: Sun Aug 13 05:04:50 2000
loop 1 done at: Sun Aug 13 05:04:52 2000
loop 0 done at: Sun Aug 13 05:04:54 2000
all DONE at: Sun Aug 13 05:04:56 2000
```

The pieces of code that sleep for 4 and 2 seconds now occur concurrently, contributing to the lower overall runtime.

The only other major change to our application is the addition of the "`sleep(6)`" call. Why is this necessary? The reason is that if we did not stop the main thread from continuing, it would proceed to the next statement, displaying "all done" and exit, killing both threads running `loop0()` and `loop1()`.

We did not have any code which told the main thread to wait for the child threads to complete before continuing. This is what we mean by threads requiring some sort of synchronization. In our case, we used another `sleep()` call as our synchronization mechanism. We used a value of 6 seconds because we know that both threads (which take 4 and 2 seconds, as you know) should have completed by the time the main thread has counted to 6.

You are probably thinking that there should be a better way of managing threads than creating that extra delay of 6 seconds in the main thread. Because of this delay, the overall runtime is no better than in our single-threaded version. Using `sleep()` for thread synchronization as we did is not reliable. What if our loops had independent and varying execution times? We may be exiting the main thread too early or too late. This is where locks come in.

Making yet another update to our code to include locks as well as getting rid of separate loop functions, we get `mtsleep2.py`, presented in Example 17.3. Running it, we see that the output is similar to `mtsleep1.py`. The only difference is that we did not have to wait the extra time for `mtsleep1.py` to conclude. By using locks, we were able to exit as soon as both threads had completed execution.

```
% mtsleep2.py
starting threads…
start loop 0 at: Sun Aug 13 16:34:41 2000
```

```
start loop 1 at: Sun Aug 13 16:34:41 2000
loop 1 done at: Sun Aug 13 16:34:43 2000
loop 0 done at: Sun Aug 13 16:34:45 2000
all DONE at: Sun Aug 13 16:34:45 2000
```

### Example 17.3. Using `thread` and Locks (`mtsleep2.py`)

Rather than using a call to `sleep()` to hold up the main thread as in `mtsleep1.py`, the use of locks makes more sense.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import thread
004 4  from time import sleep, time, ctime
005 5
006 6  loops = [ 4, 2 ]
007 7
008 8  def loop(nloop, nsec, lock):
009 9      print 'start loop', nloop, 'at:', ctime(time())
010 10     sleep(nsec)
011 11     print 'loop', nloop, 'done at:', ctime(time())
012 12     lock.release()
013 13
014 14 def main():
015 15     print 'starting threads…'
016 16     locks = []
017 17     nloops = range(len(loops))
018 18
019 19     for i in nloops:
020 20         lock = thread.allocate_lock()
021 21         lock.acquire()
022 22         locks.append(lock)
023 23
024 24     for i in nloops:
025 25         thread.start_new_thread(loop, \
026 26             (i, loops[i], locks[i]))
027 27
028 28     for i in nloops:
029 29         while locks[i].locked(): pass <$nopage>
030 30
031 31     print 'all DONE at:', ctime(time())
032 32
033 33 if __name__ == '__main__':
034 34    main()
035  <$nopage>
```

So how did we accomplish our task with locks? Let's take a look at the source code:

## Line-by-line explanation

### Lines 1–6

After the Unix start-up line, we import the `thread` module and a few familiar attributes of the `time` module. Rather than hardcoding separate functions to count to 4 and 2 seconds, we will use a single `loop()` function and place these constants in a list, `loops`.

### Lines 8–12

The `loop()` function will proxy for the now-removed `loop*()` functions from our earlier examples. We had to make some cosmetic changes to `loop()` so that it can now perform its duties using locks. The obvious changes are that we need to be told which loop number we are as well as how long to sleep for. The last piece of new information is the lock itself. Each thread will be allocated an acquired lock. When the `sleep()` time has concluded, we will release the corresponding lock, indicating to the main thread that this thread has completed.

### Lines 14–34

The bulk of the work is done here in `main()` using three separate **`for`** loops. We first create a list of locks, which we obtain using the `thread.allocate_lock()` function and acquire each lock with the `acquire()` method. Acquiring a lock has the effect of "locking the lock." Once it's locked, we add the lock to the lock list, `locks`. The next loop actually spawns the threads, invoking the `loop()` function per thread, and for each thread, provides it with the loop number, the time to sleep for, and the acquired lock for that thread. So why didn't we start the threads in the lock acquisition loop? There are several reasons: (1) we wanted to synchronize the threads, so that "all the horses started out the gate" around the same time, and (2) locks take a little bit of time to be acquired. If your thread executes "too fast," it is possible that it completes before the lock has a chance to be acquired.

It is up to each thread to unlock its lock object when it has completed execution. The final loop just sits-and-spins (pausing the main thread) until both locks have been released before continuing execution. Since we are checking each lock sequentially, we may be at the mercy of all the slower loops if they are more towards the beginning of the set of loops. In such cases, the majority of the wait time may be for the first loop(s). When that lock is released, remaining locks may have already been unlocked (meaning that corresponding threads have completed execution). The result is that the main thread will fly through those lock checks without pause. Finally, you should be well aware that the final pair of lines will execute `main()` only if we are invoking this script directly.

As hinted in the earlier Core Note, we presented the `thread` module only to introduce the reader to threaded programming. Your MT application should use higher-level modules such as the `threading` module, which we will now discuss.

## `threading` Module

We will now introduce the higher-level `threading` module which gives you not only a `Thread` class but also a wide variety of synchronization mechanisms to use to your heart's

content. Table 17.2 represents a list of all the objects which are provided for in the `threading` module.

**Table 17.2. `threading` Module Objects**

| ***`threading` Module Objects*** | ***Description*** |
|---|---|
| `Thread` | object which represents a single thread of execution |
| `Lock` | primitive lock object (same lock object as in the `thread` module) |
| `RLock` | re-entrant lock object provides ability for a single thread to (re)acquire an already-held lock (recursive locking) |
| `Condition` | condition variable object causes one thread to wait until a certain "condition" has been satisfied by another thread, such as changing of state or of some data value |
| `Event` | general version of condition variables whereby any number of threads are waiting for some event to occur and all will awaken when the event happens |
| `Semaphore` | provides a "waiting area"-like structure for threads waiting on a lock |

In this section, we will examine how to use the `Thread` class to implement threading. Since we have already covered the basics of locking, we will not cover the locking primitives here. The `Thread()` class also contains a form of synchronization, so explicit use of locking primitives is not necessary.

## `Thread` Class

There are a variety of ways you can create threads using the `Thread` class. We cover three of them here, all quite similar. Pick the one you feel most comfortable with, not to mention the most appropriate for your application and future scalability (we like choice 3 the best):

Create `Thread` instance, passing in function

Create `Thread` instance, passing in callable class instance

Subclass `Thread` and create subclass instance

### Create `Thread` instance, passing in function

In our first example, we will just instantiate `Thread`, passing in our function (and its arguments) in a manner similar to our previous examples. This function is what will be executed when we direct the thread to begin execution. Taking our `mtsleep2.py` script and tweaking it, adding the use of `Thread` objects, we have `mtsleep3.py`, shown in Example 17.4.

When we run it, we see output similar to its predecessors':

```
% mtsleep3.py
starting threads…
start loop 0 at: Sun Aug 13 18:16:38 2000
start loop 1 at: Sun Aug 13 18:16:38 2000
loop 1 done at: Sun Aug 13 18:16:40 2000
loop 0 done at: Sun Aug 13 18:16:42 2000
all DONE at: Sun Aug 13 18:16:42 2000
```

So what *did* change? Gone are the locks which we had to implement when using the `thread` module. Instead, we create a set of `Thread` objects. When each `Thread` is instantiated, we dutifully pass in the function (`target`) and arguments (`args`) and receive a `Thread` instance in return. The biggest difference between instantiating `Thread` [calling `Thread()`] and invoking `thread.start_new_thread()` is that the new thread does not begin execution right away. This is a useful synchronization feature, especially when you don't want the threads to start immediately.

### Example 17.4. Using the threading Module (`mtsleep3.py`)

The `Thread` class from the `threading` module has a `join()` method which lets the main thread wait for thread completion.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import threading
004 4  from time import sleep, time, ctime
005 5
006 6  loops = [ 4, 2 ]
007 7
008 8  def loop(nloop, nsec):
009 9      print 'start loop', nloop, 'at:', ctime(time())
010 10     sleep(nsec)
011 11     print 'loop', nloop, 'done at:', ctime(time())
012 12
013 13 def main():
014 14     print 'starting threads…'
015 15     threads = []
016 16     nloops = range(len(loops))
017 17
018 18     for i in nloops:
019 19         t = threading.Thread(target=loop,
020 20             args=(i, loops[i]))
021 21         threads.append(t)
022 22
023 23     for i in nloops:          # start threads
024 24         threads[i].start()
025 25
026 26     for i in nloops:          # wait for all
027 27         threads[i].join(       # threads to finish
028 28
029 29     print 'all DONE at:', ctime(time())
030 30
```

```
031 31 if __name__ == '__main__':
032 32     main()
033  <$nopage>
```

Once all the threads have been allocated, we let them go off to the races by invoking each thread's `start()` method, but not a moment before that. And rather than having to manage a set of locks (allocating, acquiring, releasing, checking lock state, etc.), we simply call the `join()` method for each thread. `join()` will wait until a thread terminates, or, if provided, a timeout occurs. Use of `join()` appears much cleaner than an infinite loop waiting for locks to be released (causing these locks to sometimes be known as "spin locks").

One other important aspect of `join()` is that it does not need to be called at all. Once threads are started, they will execute until their given function completes, whereby they will exit. If your main thread has things to do other than wait for threads to complete (such as other processing or waiting for new client requests), it should be all means do so. `join()` is useful only when you *want* to wait for thread completion.

### Create `Thread` instance, passing in callable class instance

A similar offshoot to passing in a function when creating a thread is to have a callable class and passing in an instance for execution—this is the more OO approach to MT programming. Such a callable class embodies an execution environment that is much more flexible than a function or choosing from a set of functions. You now have the power of a class object behind you, as opposed to a single function or a list/tuple of functions.

Adding our new class `ThreadFunc` to the code and making other slight modifications to `mtsleep3.py`, we get `mtsleep4.py`, given in Example 17.5.

If we run `mtsleep4.py`, we get the expected output:

```
% mtsleep4.py
starting threads…
start loop 0 at: Sun Aug 13 18:49:17 2000
start loop 1 at: Sun Aug 13 18:49:17 2000
loop 1 done at: Sun Aug 13 18:49:19 2000
loop 0 done at: Sun Aug 13 18:49:21 2000
all DONE at: Sun Aug 13 18:49:21 2000
```

So what are the changes this time? The addition of the `ThreadFunc` class and a minor change to instantiate the `Thread` object, which also instantiates `ThreadFunc`, our callable class. In effect, we have a double instantiation going on here. Let's take a closer look at our `ThreadFunc` class.

We want to make this class general enough to use with other functions besides our `loop()` function, so we added some new infrastructure, such as having this class hold the arguments for the function, the function itself, and also a function name string. The constructor `__init__()` just sets all the values.

### Example 17.5. Using Callable classes (`mtsleep4.py`)

In this example we pass in a callable class (instance) as opposed to just a function. It presents more of an OO approach than `mtsleep3.py`.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import threading
004 4  from time import sleep, time, ctime
005 5
006 6  loops = [ 4, 2 ]
007 7
008 8  class ThreadFunc:
009 9
010 10     def __init__(self, func, args, name=''):
011 11         self.name = name
012 12         self.func = func
013 13         self.args = args
014 14
015 15     def __call__(self):
016 16         apply(self.func, self.args)
017 17
018 18 def loop(nloop, nsec):
019 19     print 'start loop', nloop, 'at:', ctime(time())
020 20     sleep(nsec)
021 21     print 'loop', nloop, 'done at:', ctime(time())
022 22
023 23 def main():
024 24     print 'starting threads…'
025 25     threads = []
026 26     nloops = range(len(loops))
027 27
028 28     for i in nloops: # create all threads
029 29         t = threading.Thread( \
030 30             target=ThreadFunc(loop, (i, loops[i]),
031 31             loop.__name__))
032 32         threads.append(t)
033 33
034 34     for i in nloops: # start all threads
035 35         threads[i].start()
036 36
037 37     for i in nloops: # wait for completion
038 38         threads[i].join()
039 39
040 40     print 'all DONE at:', ctime(time())
041 41
042 42 if __name__ == '__main__':
043 43     main()
044  <$nopage>
```

When the `Thread` code calls our `ThreadFunc` object when a new thread is created, it will invoke the `__call__()` special method. Because we already have our set of arguments, we do not need to pass it to the `Thread()` constructor, but do have to use `apply()` in our code now because we have an argument tuple. Those of you who have Python 1.6 and higher can use the new function invocation syntax described in Section 11.6.3 instead of using `apply()` on line 17:

```
self.res = self.func(*self.args)
```

### Subclass `Thread` and create subclass instance

The final introductory example involves subclassing `Thread()`, which turns out to be extremely similar to creating a callable class as in the previous example. Subclassing is a bit easier to read when you are creating your threads (lines 28–29). We will present the code for `mtsleep5.py` in Example 17.6 as well as the output obtained from its execution, and leave it as an exercise for the reader to compare `mtsleep5.py` to `mtsleep4.py`.

Here is the output for `mtsleep5.py`, again, just what we expected:

```
% mtsleep5.py
starting threads…
start loop 0 at: Sun Aug 13 19:14:26 2000
start loop 1 at: Sun Aug 13 19:14:26 2000
loop 1 done at: Sun Aug 13 19:14:28 2000
loop 0 done at: Sun Aug 13 19:14:30 2000
all DONE at: Sun Aug 13 19:14:30 2000
```

While the reader compares the source between the `mtsleep4` and `mtsleep5` modules, we want to point out the most significant changes: (1) our `MyThread` subclass constructor must first invoke the base class constructor (line 9), and (2) the former special method `__call__()` must be called `run()` in the subclass.

We now modify our `MyThread` class with some diagnostic output and store it in a separate module called `myThread` (see Example 17.7) and import this class for the upcoming examples. Rather than simply calling `apply()` to run our functions, we also save the result to instance attribute `self.res`, and create a new method to retrieve that value, `getResult()`.

### Example 17.6. Subclassing `Thread` (`mtsleep5.py`)

Rather than instantiating the `Thread` class, we subclass it. This gives us more flexibility in customizing our threading objects and simplifies the thread creation call.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import threading
004 4  from time import sleep, time, ctime
005 5
006 6  loops = ( 4, 2 )
007 7
008 8  class MyThread(threading.Thread):
009 9      def __init__(self, func, args, name=''):
010 10         threading.Thread.__init__(self)
011 11         self.name = name
012 12         self.func = func
013 13         self.args = args
014 14
015 15     def run(self):
016 16         apply(self.func, self.args)
017 17
018 18 def loop(nloop, nsec):
019 19     print 'start loop', nloop, 'at:', ctime(time())
020 20     sleep(nsec)
021 21     print 'loop', nloop, 'done at:', ctime(time())
022 22
023 23 def main():
024 24     print 'starting threads…'
025 25     threads = []
026 26     nloops = range(len(loops))
027 27
028 28     for i in nloops:
029 29         t = MyThread(loop, (i, loops[i]), \
030 30             loop.__name__)
031 31         threads.append(t)
032 32
033 33     for i in nloops:
034 34         threads[i].start()
035 35
036 36     for i in nloops:
037 37         threads[i].join()
038 38
039 39     print 'all DONE at:', ctime(time())'
040 40
041 41 if __name__ == '__main__':
042 42     main()
043  <$nopage>
```

**Example 17.7. `MyThread` Subclass of Thread (`myThread.py`)**

To generalize our subclass of `Thread` from `mtsleep5.py`, we move the subclass to a separate module and add a `getResult()` method for callables which produce return values.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import threading
```

```
004 4  from time import time, ctime
005 5
006 6  class MyThread(threading.Thread):
007 7      def __init__(self, func, args, name=''):
008 8          threading.Thread.__init__(self)
009 9          self.name = name
010 10         self.func = func
011 11         self.args = args
012 12
013 13     def getResult(self):
014 14         return self.res
015 15
016 16     def run(self):
017 17         print 'starting', self.name, 'at:', \
018 18             ctime(time())
019 19         self.res = apply(self.func, self.args)
020 20         print self.name, 'finished at:', \
021 21             ctime(time())
022  <$nopage>
```

## Fibonacci and factorial... take 2, plus summation

The `mtfacfib.py` script, given in Example 17.8, compares execution of the recursive Fibonacci, factorial, and summation functions. This script runs all three functions in a single-threaded manner, then performs the same task using threads to illustrate one of the advantages of having a threading environment.

### Example 17.8. Fibonacci, Factorial, Summation (`mtfacfib.py`)

In this MT application, we execute 3 separate recursive functions—first in a single-threaded fashion, followed by the alternative with multiple threads.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from myThread import MyThread
004 4  from time import time, ctime, sleep
005 5
006 6  def fib(x):
007 7      sleep(0.005)
008 8      if x < 2: return 1
009 9      return (fib(x-2) + fib(x-1))
010 10
011 11 def fac(x):
012 12     sleep(0.1)
013 13     if x < 2: return 1
014 14     return (x * fac(x-1))
015 15
016 16 def sum(x):
017 17     sleep(0.1)
018 18     if x < 2: return 1
019 19     return (x + sum(x-1))
020 20
021 21 funcs = [fib, fac, sum]
```

```
022 22 n = 12
023 23
024 24 def main():
025 25     nfuncs = range(len(funcs))
026 26
027 27     print '*** SINGLE THREAD'
028 28     for i in nfuncs:
029 29         print 'starting', funcs[i].__name__, 'at:', \
030 30             ctime(time())
031 31         print funcs[i](n)
032 32         print funcs[i].__name__, 'finished at:', \
033 33             ctime(time())
034 34
035 35     print '\n*** MULTIPLE THREADS'
036 36     threads = []
037 37     for i in nfuncs:
038 38         t = MyThread(funcs[i], (n,),
039 39             funcs[i].__name__)
040 40         threads.append(t)
041 41
042 42     for i in nfuncs:
043 43         threads[i].start()
044 44
045 45     for i in nfuncs:
046 46         threads[i].join()
047 47         print threads[i].getResult()
048 48
049 49     print 'all DONE'
050 50
051 51 if __name__ == '__main__':
052 52     main()
053  <$nopage>
```

Running in single-threaded mode simply involves calling the functions one at a time and displaying the corresponding the results right after the function call.

When running in multithreaded mode, we do not display the result right away. Because we want to keep our `MyThread` class as general as possible (being able to execute callables which do and do not produce output), we wait until the end to call the `getResult()` method to finally show you the return values of each function call.

Because these functions execute so quickly (well, maybe except for the Fibonacci function), you will noticed that we had to add calls to `sleep()` to each function to slow things down so that we can see how threading may improve performance, if indeed the actual work had varying execution times—you certainly wouldn't pad your work with calls to `sleep()`. Anyway, here is the output:

```
% mtfacfib.py
*** SINGLE THREAD
starting fib at: Sun Jun 18 19:52:20 2000
233
fib finished at: Sun Jun 18 19:52:24 2000
starting fac at: Sun Jun 18 19:52:24 2000
```

```
479001600
fac finished at: Sun Jun 18 19:52:26 2000
starting sum at: Sun Jun 18 19:52:26 2000
78
sum finished at: Sun Jun 18 19:52:27 2000

*** MULTIPLE THREADS
starting fib at: Sun Jun 18 19:52:27 2000
starting fac at: Sun Jun 18 19:52:27 2000
starting sum at: Sun Jun 18 19:52:27 2000
fac finished at: Sun Jun 18 19:52:28 2000
sum finished at: Sun Jun 18 19:52:28 2000
fib finished at: Sun Jun 18 19:52:31 2000
233
479001600
78
all DONE
```

## Producer-Consumer Problem and the `Queue` Module

The final example illustrates the producer-consumer scenario where a producer of goods or services creates goods and places it in a data structure such as a queue. The amount of time between producing goods is non-deterministic, as is the consumer consuming the goods produced by the producer.

We use the `Queue` module to provide an interthread communication mechanism which allows threads to share data with each other. In particular, we create a queue for the producer (thread) to place new goods into and where the consumer (thread) can consume goods from.

In particular, we will use the following attributes from the `Queue` module (see Table 17.3).

**Table 17.3. Common `Queue` Module Attributes**

| *Function/Method* | *Description* |
|---|---|
| **`Queue` *Module Function*** | |
| `queue(`*`size`*`)` | creates a `Queue` object of given *`size`* |
| **`Queue` *Object Methods*** | |
| `qsize()` | returns queue size (approximate, since queue may be getting updated by other threads) |
| `empty()` | returns 1 if queue empty, 0 otherwise |
| `full()` | returns 1 if queue full, 0 otherwise |
| `put(`*`item`*`, `*`block`*`=0)` | puts *`item`* in queue, if *`block`* given (not 0), block until room is available |
| `get(`*`block`*`=0)` | gets *`item`* from queue, if *`block`* given (not 0), block until an |

| | item is available |
|---|---|

Without further ado, we present the code for `prodcons.py`, shown in Example 17.9.

### Example 17.9. Producer-Consumer Problem (`prodcons.py`)

We feature an implementation of the Producer–Consumer problem using `Queue` objects and a random number of goods produced (and consumed). The producer and consumer are individually—and concurrently—executing threads.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from random import randint
004 4  from time import time, ctime, sleep
005 5  from Queue import Queue
006 6  from myThread import MyThread
007 7
008 8  def writeQ(queue):
009 9      print 'producing object for Q…',
010 10     queue.put('xxx', 1)
011 11     print "size now", queue.qsize()
012 12
013 13 def readQ(queue):
014 14     val = queue.get(1)
015 15 print 'consumed object from Q… size now', \
016 16             queue.qsize()
017 17
018 18 def writer(queue, loops):
019 19     for i in range(loops):
020 20         writeQ(queue)
021 21         sleep(randint(1, 3))
022 22
023 23 def reader(queue, loops):
024 24     for i in range(loops):
025 25         readQ(queue)
026 26         sleep(randint(2, 5))
027 27
028 28 funcs = [writer, reader]
029 29 nfuncs = range(len(funcs))
030 30
031 31 def main():
032 32     nloops = randint(2, 5)
033 33     q = Queue(32)
034 34
035 35     threads = []
036 36     for i in nfuncs:
037 37         t = MyThread(funcs[i], (q, nloops), \
038 38             funcs[i].__name__)
039 39         threads.append(t)
040 40
041 41     for i in nfuncs:
042 42         threads[i].start()
043 43
```

```
044 44     for i in nfuncs:
045 45         threads[i].join()
046 46
047 47     print 'all DONE'
048 48
049 49 if __name__ == '__main__':
050 50     main()
051  <$nopage>
```

Here is the output from one execution of this script:

```
% prodcons.py
starting writer at: Sun Jun 18 20:27:07 2000
producing object for Q… size now 1
starting reader at: Sun Jun 18 20:27:07 2000
consumed object from Q… size now 0
producing object for Q… size now 1
consumed object from Q… size now 0
producing object for Q… size now 1
producing object for Q… size now 2
producing object for Q… size now 3
consumed object from Q… size now 2
consumed object from Q… size now 1
writer finished at: Sun Jun 18 20:27:17 2000
consumed object from Q… size now 0
reader finished at: Sun Jun 18 20:27:25 2000
all DONE
```

As you can see, the producer and consumer do not necessarily alternate in execution. (Thank goodness for random numbers!) Seriously though, real life is generally random and non-deterministic.

### Line-by-line explanation

### Lines 1–6

In this module, we will use the `Queue.Queue` object as well as our thread class `myThread.MyThread` which we gave in Example 17.7. We will use `random.randint()` to make production and consumption somewhat varied, and also grab the usual suspects from the time module.

### Lines 8–16

The `writeQ()` and `readQ()` functions each have a specific purpose, to place an object in the queue—we are using the string `'xxx'` for example—and to consume a queued object, respectively. Notice that we are producing one object and reading one object each time.

### Lines 18–26

The `writer()` is going to run as a single thread who sole purpose is to produce an item for the queue, wait for a bit, then do it again, up to the specified number of times, chosen

randomly per script execution. The `reader()` will do likewise, with the exception of consuming an item, of course.

You will notice that the random number of seconds that the writer sleeps is in general shorter than the amount of time the reader sleeps. This is to discourage the reader from trying to take items from an empty queue. By giving the writer a shorter time period of waiting, it is more likely that there will already be an object for the reader to consume by the time their turn rolls around again.

#### Lines 28–29

These are just setup lines to set the total number of threads that are to be spawned and executed.

#### Lines 31–47

Finally, our `main()` function, which should look quite similar to the `main()` in all of the other scripts in this chapter. We create the appropriate threads and send them on their way, finishing up when both threads have concluded execution.

We infer from this example that a program that has multiple tasks to perform can be organized to use separate threads for each of the tasks. This can result in a much cleaner program design than a single threaded program that attempts to do all of the tasks.

In this chapter, we illustrated how a single-threaded process may limit an application's performance. In particular, programs with independent, non-deterministic, and non-causal tasks which execute sequentially can be improved by division into separate tasks executed by individual threads. Not all applications may benefit from multithreading and its overheads, but now you are more cognizant of Python's threading capability enough to use this tool to your advantage when appropriate.

## Exercises

*Processes vs. Threads.* What are the differences between processes and threads?

*Threads.* If multiple threads are running on a single CPU system, how do they share the CPU?

*Threads.* Do you think anything significant happens if you have multiple threads on a multiple CPU system? How do you think multiple threads run on these systems?

*Threads and Files.* Update your solution to Exercise 9-19 which obtains a byte value and a file name, displaying the number of times that byte appears in the file. Let's suppose this is a really big file. Multiple readers in a file is acceptable, so create multiple threads that count in different parts of the file so that each thread is responsible for a certain part of the file. Collate the data from each thread and provide the summed-up result. Use your

`timeit()` code to time both the single threaded version and your new multithreaded version and say something about the performance improvement.

*Threads, Files, and Regular Expressions.* You have a very large mailbox file—if you don't have one, put all of your e-mail messages together into a single text file. Your job is to take the regular expressions you designed in Chapter 15 that recognizes e-mail addresses and Web site URLs, and use them to convert all e-mail addresses and URLs in this large file into live links so that when the new file is saved as a `.html` (or `.htm`)file, will show up in a Web browser as live and clickable. Use threads to segregate the conversion process across the large text file and collate the results into a single new `.html` file. Test the results on your Web browser to ensure the links are indeed working.

*Threads and Networking.* Your solution to the chat service application in the previous chapter (Exercises 16-7 to 16-10) may have required you to use heavyweight threads or processes as part of your solution. Convert that to be multithreaded `code.`

**Threads and Web Programming.* The `Crawler` in Example 19.1 is a single-threaded application that downloads Web pages that would benefit from MT programming. Update `crawl.py` (you could call it `mtcrawl.py`) such that independent threads are used to download pages. Be sure to use some kind of locking mechanism to prevent conflicting access to the links queue.

# Chapter 18. GUI Programming with Tkinter

In this chapter, we will give you a quick introduction to Graphical User Interface (GUI) programming with Tkinter, Python's Tk graphics toolkit. GUI development has enough material to warrant its own text (and it has!), but if you are somewhat new or want to learn more about it, or if you want to see how it's done in Python, then this chapter is for you. We present four simple examples and one intermediate example, and will defer a more complete tour of Tkinter to texts devoted purely to GUI programming in Python.

## Introduction

### What Are Tcl, Tk, and Tkinter?

Tkinter is Python's default graphical user interface library. It is based on the Tk toolkit, originally designed for the Tool Command Language (TCL). Due to Tk's popularity, it has been ported to a variety of other scripting languages, including Perl (Perl/Tk) and Python (Tkinter). With the GUI development portability and flexibility of Tk, along with the simplicity of scripting language integrated with the power of systems language, you are given the tools to rapidly design and implement a wide variety of commercial-quality GUI applications.

If you are new to GUI programming, you will be pleasantly surprised at how easy it is. You will also find that Python, along with Tkinter, provide a fast and exciting way to build applications that are fun (and perhaps useful) and that would have taken much longer if you had to program directly in C/C++ with the native windowing system's libraries. Once you have designed the application and the look-and-feel that goes along with your program, you will use basic building blocks known as widgets to piece together the desired components, and finally, to attach functionality to "make it real."

If you are an "old-hat" at using Tk, either with Tcl or Perl, you will find Python a refreshing way to program GUIs, on top of that, it provides an even faster rapid prototyping system for building GUIs. Remember that you also have Python's system-accessibility, networking functionality, XML, numerical and visual processing, database access, and all the other standard library and third-party extension modules.

Once you get Tkinter up on your system, it will take less than 15 minutes to get your first GUI app running.

### Getting Tkinter Installed and Working

Like threading, Tkinter is not necessarily turned on by default on your system. You can tell whether Tkinter is available for your Python interpreter by attempting to import the `Tkinter` module. If Tkinter is available, then no errors occur:

```
>>> import Tkinter
>>>
```

If your Python interpreter was not compiled with Tkinter enabled, the module import fails:

```
>>> import Tkinter
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "/usr/lib/python1.5/lib-tk/Tkinter.py", line 8, in ?
    import _tkinter # If this fails your Python may not
be configured for Tk
ImportError: No module named _tkinter
```

You may have to recompile your Python interpreter to get access to Tkinter. This usually involves editing the `Modules/Setup` file and enabling all the correct settings to compile your Python interpreter with hooks to Tkinter or choosing to have Tk installed on your system. Check the `README` file for your Python distribution for specific instructions on getting Tkinter to compile on your system. Be sure, after your compilation, that you start the *new* Python interpreter you just created; otherwise, it will act just like your old one without Tkinter (and in fact, it *is* your old one).

### Client-Server Architecture—Take 2

In the earlier chapter on network programming, we introduced the notion of client-server computing. A windowing system is another example of a software server. These run on a machine with an attached display, such as a monitor of some sort. There are clients too—programs which require a windowing environment to execute, also known as GUI applications. Such applications cannot run without a windows system.

The architecture becomes even more interesting when networking comes into play. Usually when a GUI application is executed, it displays to the machine that it started on (via the windowing server), but it is possible in some networked windowing environments, such as the X Windows system on Unix, to choose another machine's window server to display to. In such situations, you can be running a GUI program on one machine, but have it displayed at another!

## Tkinter and Python Programming

### `Tkinter` Module: Adding Tk to your Applications

So what do you need to do to have Tkinter as part of your application? Well, first of all, it is not necessary to have an application already. You can create a pure graphical user interface if you want, but it probably isn't too useful without some underlying software that does something interesting.

There are basically five main steps that are required to get your GUI up-and-running:

Import the `Tkinter` module (or `from Tkinter import *`).

Create a top-level windowing object which contains your entire GUI application.

Built all your GUI components (and functionality) on top (or "inside") of your top-level windowing object).

Connect these GUI components to the underlying application code.

Enter the main event loop.

The first step is trivial: All GUIs which use Tkinter must import the `Tkinter` module. Getting access to Tkinter is the first step (see the previous Section 18.1.2).

## Introduction to GUI Programming

Before going to the examples, we will give you a brief introduction to GUI application development in general. This will give you some of the background you need to move forward.

Setting up a GUI application is similar to an artist's producing a painting. Conventionally, there is a single canvas onto which the artist must put all the work. The way it works is like this: You start with a clean slate, a "top-level" windowing object on which you build the rest of your components. Think of it as a foundation to a house or the easel for an artist. In other words, you have to pour the concrete or set up your easel before putting together the actual structure or canvas on top of it. In Tkinter, this foundation is known as the top-level window object.

In GUI programming, a top-level root windowing object contains all of the little windowing objects that will be part of your complete GUI application. These can be text labels, buttons, list boxes, etc. These individual little GUI components are known as *widgets*. So when we say create a top-level window, we just mean that you need such a thing as a place where you put all your widgets. In Python, this would typically look like the line below :

```
top = Tkinter.Tk() # or just Tk() with "from Tkinter import *"
```

The object returned by `Tkinter.Tk()` is usually referred to as the *root object,* hence the reason why some applications use `root` rather than `top` to indicate as such. Top-level windows are those which show up standalone as part of your application, and, yes, you may have more than one top-level window for your GUI, but only one of them should be your root window. You may choose to completely design all your widgets first, then add the real functionality, or do a little of this and a little of that along the way. (This means mixing and matching steps 3 and 4 from our list above.)

Widgets may be standalone or be containers. If a widget "contains" other widgets, it is considered the *parent* of those widgets. Accordingly, if a widget is "contained" in another widget, it's considered a *child* of the parent, the parent being the next immediate enclosing container widget.

Usually, widgets have some associated behaviors, such as when a button is pressed, or text is filled into a text field. These types of user behaviors are called *events,* and the actions that the GUI takes to respond to such events are known as *callbacks.*

Actions may include the actual button press (and release), mouse movement, hitting the RETURN or Enter key, etc. All of these are known to the system literally as *events.* The entire system of events which occurs from the beginning to the end of a GUI application is what drives it. This is known as event-driven processing.

One example of an event with a callback is a simple mouse move. Let's say the mouse pointer is sitting somewhere on top of your GUI application. If the mouse is moved to another part of your application, something has to cause the movement of the mouse on your screen so that it *looks* as if it is moving to another location. These are mouse move events that the system must process to give you the illusion (and reality) that your mouse is moving across the window. When you release the mouse, there are no more events to process, so everything just sits there quietly on the screen again.

The event-driven processing nature of GUIs fits right in with client-server architecture. When you start a GUI application, it must perform some setup procedures to prepare for the core execution, just as when a network server has to allocate a socket and bind it to a local address. The GUI application must establish all the GUI components, then draw (a.k.a. render or paint) them to the screen. Tk has a couple of geometry managers which help position the widget in the right place; the main one which you will use is called the *packer.* Once the packer has determined the sizes and alignments of all your widgets, it will then place them on the screen for you.

When all of the widgets, including the top-level window finally appear on your screen, your GUI application then enters a "server-like" infinite loop. This infinite loop involves waiting for a GUI event, processing it, then going back to wait for the next event.

The final step we described above says to enter the main loop once all the widgets are ready. This is the "server" infinite loop we have been referring to. In Tkinter, the code that does this is:

```
Tkinter.mainloop()
```

This is normally the last piece of sequential code your program runs. When the main loop is entered, the GUI takes over execution from there. All other action is via callbacks, even exiting your application. When you pull down the File menu to click on the Exit menu option or close the window directly, a callback must be invoked to end your GUI application.

## Top-level window: `Tkinter.Tk()`

We mentioned above that all main widgets are built into the top-level window object. This object is created by the `Tk` class in Tkinter and is created via the normal instantiation:

```
>>> import Tkinter
>>> top = Tkinter.Tk()
```

Within this window, you place individual widgets or multiple-component pieces together to form your GUI. So what kinds of widgets are there? We will now introduce the Tk widgets.

## Tk Widgets

There are currently 15 types of widgets in Tk. We present these widgets as well as a brief description in Table 18.1.

We won't go over the Tk widgets in detail as there is plenty of good documentation available on them, either from the Tkinter topics page at the main Python Web site or the abundant number of Tcl/Tk printed and online resources (some of which are available in the Appendix). However, we will present several simple examples to help you get started.

**NOTE**

GUI development really takes advantage of default arguments in Python because there are numerous default actions in Tkinter widgets. Unless you know every single option available to you for every single widget you are using, it's best to start out by setting only the parameters you are aware of and letting the system handle the rest. These defaults were chosen carefully. If you do not provide these values, do not worry about your applications appearing odd on the screen. They were created with an optimized set of default arguments as a general rule, and only when you know how to exactly customize your widgets should you use values other than the default.

| Table 18.1. Tk Widgets | |
|---|---|
| ***Widgets*** | ***Description*** |
| `Button` | similar to a `Label` but provides additional functionality for mouse overs, presses, and releases as well as keyboard activity/events |
| `Canvas` | provides ability to draw shapes (lines, ovals, polygons, rectangles); can contain images or bitmaps |
| `Checkbutton` | set of boxes of which any number can be "checked" (similar to HTML `checkbox` input) |
| `Entry` | single-line text field with which to collect keyboard input (similar to HTML text input) |
| `Frame` | pure container for other widgets |
| `Label` | used to contain text or images |
| `Listbox` | presents user list of choices to pick from |
| `Menu` | actual list of choices "hanging" from a `Menubutton` that the user can choose from |
| `Menubutton` | provides infrastructure to contain menus (pulldown, cascading, etc.) |
| `Message` | similar to a `Label,` but displays multi-line text |
| `Radiobutton` | set of buttons of which only one can be "pressed" (similar to HTML `radio` input) |
| `Scale` | linear "slider" widget providing an exact value at current setting; with defined starting and ending values |
| `Scrollbar` | provides scrolling functionality to supporting widgets, i.e., `Text, Canvas, Listbox,` and `Entry` |
| `Text` | multi-line text field with which to collect (or display) text from user (similar to HTML `textarea`) |
| `Toplevel` | similar to a `Frame,` but provides a separate window container |

## Tkinter Examples

### `Label` Widget

In Example 18.1, we present `tkhello1.py,` the Tkinter version of "Hello World!" In particular, it shows you how a Tkinter application is set up and highlights the `Label` widget

#### Example 18.1. `Label` Widget Demo (`tkhello1.py`)

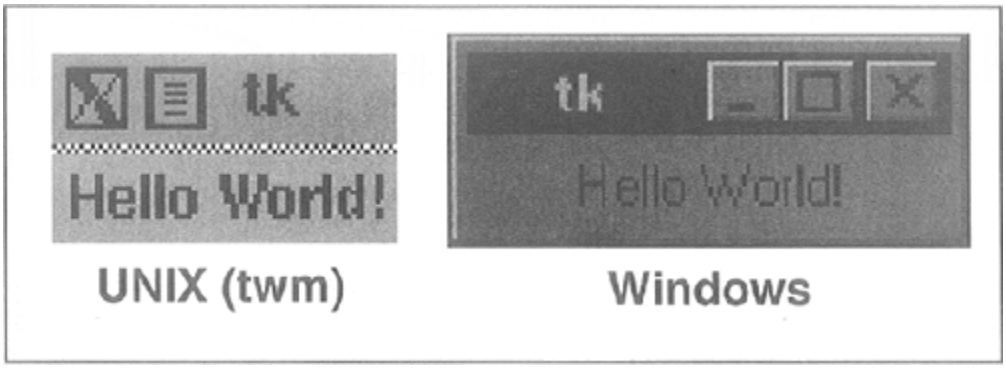
Our first Tkinter example is… what else? "Hello World!" In particular, we introduce our first widget, the `Label.`

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import Tkinter
004 4
005 5  top = Tkinter.Tk()
```

```
006 6   label = Tkinter.Label(top, text='Hello World!')
007 7   label.pack()
008 8   Tkinter.mainloop()
009  <$nopage>
```

In the first line, we create our top-level window. That is followed by our `Label` widget containing the all-too-famous string. We instruct the packer to manage and display our widget, and finally call `mainloop()` to run our GUI application. Figure18-1 shows what you will see when you run this GUI application.

**Figure 18-1. Tkinter `Label` Widget (`tkhello1.py`)**




## `Button` Widget

The next example is pretty much the same as the first. However, instead of a simple text label, we will create a button instead. In Example 18.2 is the source code for `tkhello2.py`:

**Example 18.2. `Button`Widget Demo (`tkhello2.py`)**

This example is exactly the same as `tkhello1.py` except that rather than using a `Label` widget, we create a `Button` widget.

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   import Tkinter
004 4
005 5   top = Tkinter.Tk()
006 6   quit = Tkinter.Button(top, text='Hello World!',
007 7       command=top.quit)
008 8   quit.pack()
009 9   Tkinter.mainloop()
010  <$nopage>
```

The first few lines are identical. Things differ only when we create the `Button` widget. Our button has one additional parameter, the `Tkinter.quit()` method. This installs a callback to our button so that if it is pressed (and released), the entire application will exit.

The final two lines are the usual `pack()` and entering of the `mainloop()`. This simple button application is shown in Figure18-2.

**Figure 18-2. Tkinter `Button` Widget (`tkhello2.py`)**


## `Label` and `Button` Widgets

We combine `tkhello1.py` and `tkhello2.py` into `tkhello3.py`, a script which has both a label and a button. In addition, we are providing more parameters now than before when we were comfortable using all the default arguments which are automatically set for us. The source for `tkhello3.py` is given in Example 18.3.

### Example 18.3. `Label` and `Button` Widget Demo (`tkhello3.py`)

This example features both a `Label` and a `Button` widget. Rather than primarily using default arguments when creating the widget, we are able to specify more now that we know more about `Button` widgets and how to configure them.

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   import Tkinter
004 4   top = Tkinter.Tk()
005 5
006 6   hello = Tkinter.Label(top, text='Hello World!')
007 7   hello.pack()
008 8
009 9   quit = Tkinter.Button(top, text='QUIT',
010 10      command=top.quit, bg='red', fg='white')
011 11  quit.pack(fill=Tkinter.X, expand=1)
012 12
013 13  Tkinter.mainloop()
014  <$nopage>
```

Besides additional parameters for the widgets, we also see some arguments for the packer. The `fill` parameter tells the packer to let the QUIT button take up the rest of the horizontal real estate, and the `expand` parameter directs the packer to visually fill out the entire horizontal landscape, stretching the button to the left and right sides of the window.

As you can see in Figure18-3, without any other instructions to the packer, the widgets are placed vertically (on top of each other). Horizontal placement requires creating a new `Frame` object with which to add the buttons. That frame will take the place of the parent object as a single child object (see the buttons in the `listdir.py` module, Example 18.5 in Section 18.3.5).

**Figure 18-3. Tkinter `Label` and `Button` Widgets (`tkhello3.py`)**


## `Label, Button,` and `Scale` Widgets

Our final trivial example, `tkhello4.py,` involves the addition of a `Scale` widget. In particular, the `Scale` is used to interact with the `Label` widget. The `Scale` slider is a tool which controls the size of the text font in the `Label` widget. The greater the slider position, the larger the font, and the same goes for a lesser position, meaning a smaller font. The code for `tkhello4.py` is given in Example 18.4.

New features of this script include a `resize()` callback function (lines 5–7), which is attached to the `Scale.` This is the code that is activated when the slider on the `Scale` is moved, resizing the size of the text in the `Label.`

We also define the size (250×150) of the top-level window (line 10). The final difference between this script and the first three is that we import the attributes from the `Tkinter` module into our namespace with "`from Tkinter import *.`" This is mainly due to the fact that this application is larger and involves a large number of references to Tkinter attributes, which would otherwise require their fully-qualified names. The code is shortened a bit and perhaps may not wrap as many lines without importing all the attributes locally.

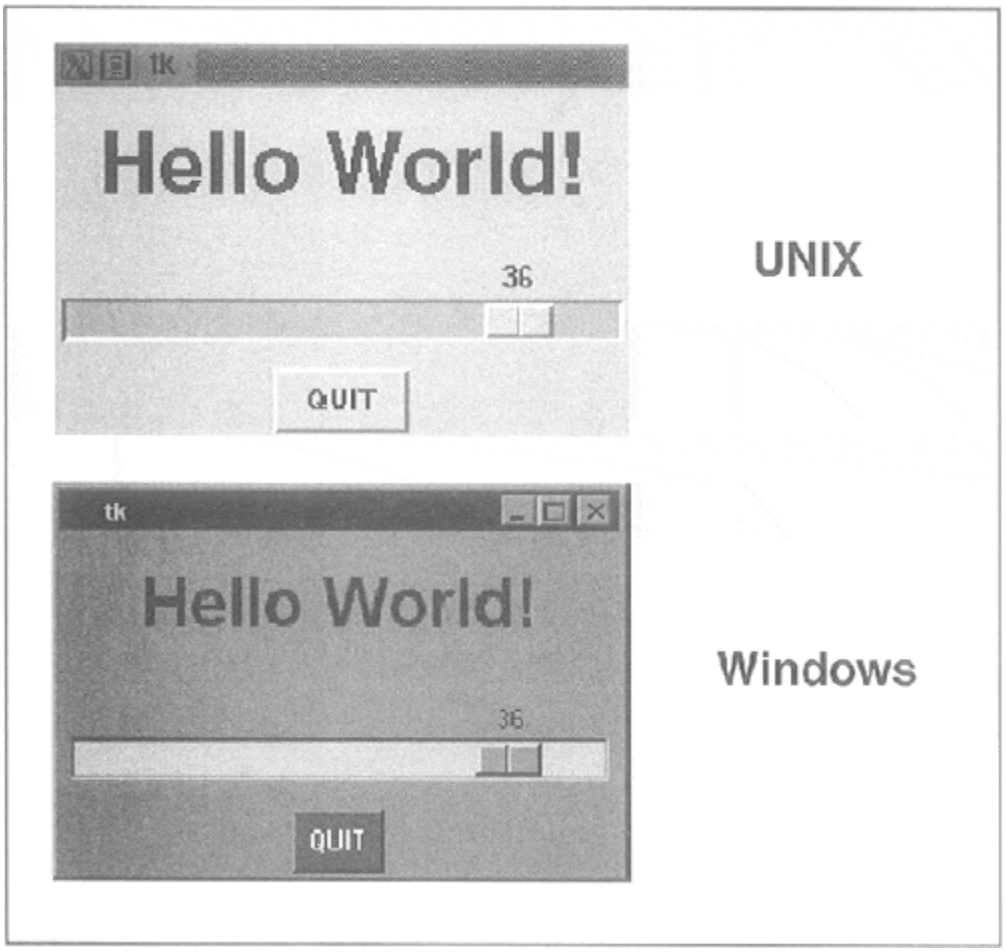
**Example 18.4. `Label, Button,` and `Scale` Demo (`tkhello4.py`)**

Our final introductory widget example introduces the ***Scale*** widget and highlights how widgets can "communicate" with each other using callbacks [such as ***resize()***]. The text in the ***Label*** widget is affected by actions taken on the ***Scale*** widget.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  from Tkinter import * <$nopage>
004 4
005 5  def resize(ev=None):
006 6      label.config(font='Helvetica -%d bold' % \
007 7          scale.get())
008 8
009 9  top = Tk()
010 10 top.geometry('250×150')
011 11
012 12 label = Label(top, text='Hello World!',
013 13     font='Helvetica -12 bold')
014 14 label.pack(fill=Y, expand=1)
015 15
016 16 scale = Scale(top, from_=10, to=40,
017 17     orient=HORIZONTAL, command=resize)
018 18 scale.set(12)
019 19 scale.pack(fill=X, expand=1)
020 20
021 21 quit = Button(top, text='QUIT',
022 22     command=top.quit, activeforeground='white',
023 23     activebackground='red')
024 24 quit.pack()
025 25
026 26 mainloop()
027  <$nopage>
```

As you can see from Figure18-4, both the slider mechanism as well as the current set value show up in the main part of the window.

**Figure 18-4. Tkinter `Label`, `Button`, and `Scale` Widgets (`tkhello4.py`)**




## Intermediate Tkinter Example

We conclude this section with a larger example, `listdir.py`. This application is a directory tree traversal tool. It starts in the current directory and provides a file listing. Double-clicking on any other directory in the list causes the tool to change to the new directory as well as replace the original file listing with the files from the new directory. The source code is given below as Example 18.5.

### Example 18.5. File System Traversal GUI (`listdir.py`)

This slightly more advanced GUI expands on the use of widgets, adding listboxes, text entry fields, and scrollbars to our repertoire. There are also a good number of callbacks such as mouse clicks, key presses, and scrollbar action.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import os
004 4  from time import sleep
005 5  from Tkinter import * <$nopage>
006 6
```

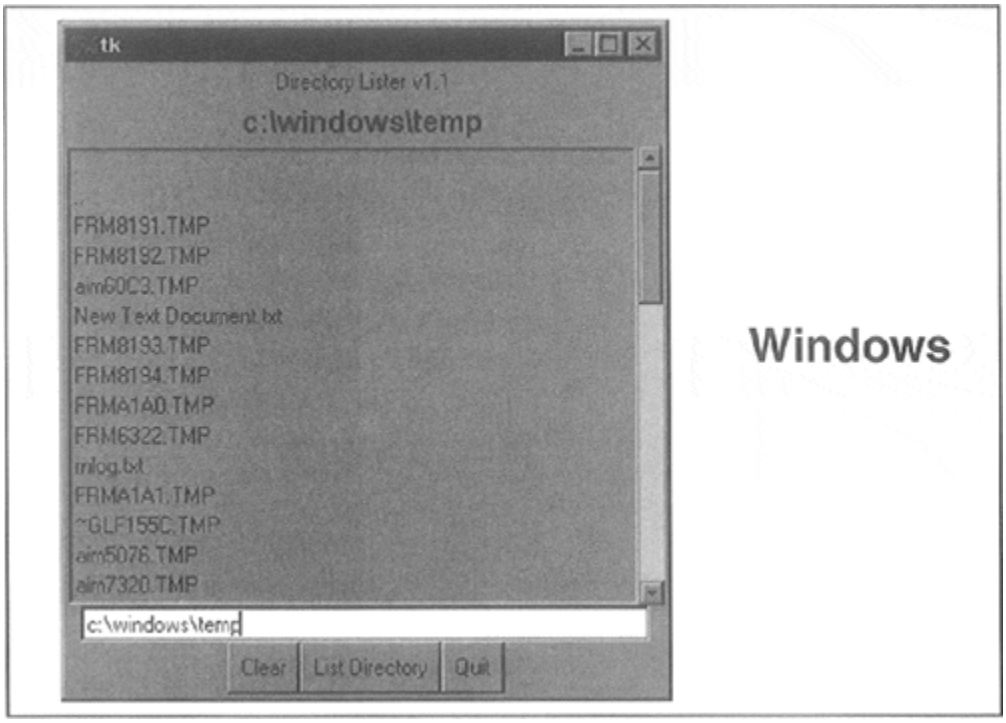
```
007 7  class DirList:
008 8
009 9       def __init__(self, initdir=None):
010 10          self.top = Tk()
011 11          self.label = Label(self.top,
012 12              text='Directory Lister' + ' v1.1')
013 13          self.label.pack()
014 14
015 15          self.cwd=StringVar(self.top)
016 16
017 17          self.dirl = Label(self.top, fg='blue',
018 18              font=('Helvetica', 12, 'bold'))
019 19          self.dirl.pack()
020 20
021 21          self.dirfm = Frame(self.top)
022 22          self.dirsb = Scrollbar(self.dirfm)
023 23          self.dirsb.pack(side=RIGHT, fill=Y)
024 24          self.dirs = Listbox(self.dirfm, height=15,
025 25              width=50, yscrollcommand=self.dirsb.set)
026 26          self.dirs.bind('<Double-1>', self.setDirAndGo)
027 27          self.dirsb.config(command=self.dirs.yview)
028 28          self.dirs.pack(side=LEFT, fill=BOTH)
029 29          self.dirfm.pack()
030 30
031 31          self.dirn = Entry(self.top, width=50,
032 32              textvariable=self.cwd)
033 33          self.dirn.bind('<Return>', self.doLS)
034 34          self.dirn.pack()
035 35
036 36          self.bfm = Frame(self.top)
037 37          self.clr = Button(self.bfm, text='Clear',
038 38              command=self.clrDir,
039 39              activeforeground='white',
040 40              activebackground='blue')
041 41          self.ls = Button(self.bfm,
042 42              text='List.Directory',
043 43              command=self.doLS,
044 44              activeforeground='white',
045 45              activebackground='green')
046 46          self.quit = Button(self.bfm, text='Quit',
047 47              command=self.top.quit,
048 48              activeforeground='white',
049 49              activebackground='red')
050 50          self.clr.pack(side=LEFT)
051 51          self.ls.pack(side=LEFT)
052 52          self.quit.pack(side=LEFT)
053 53          self.bfm.pack()
054 54
055 55          if initdir:
056 56              self.cwd.set(os.curdir)
057 57              self.doLS()
058 58
059 59      def clrDir(self, ev=None):
060 60          self.cwd.set('')
061 61
062 62      def setDirAndGo(self, ev=None):
063 63          self.last = self.cwd.get()
```

```
064 64          self.dirs.config(selectbackground='red')
065 65          check = self.dirs.get(self.dirs.curselection())
066 66          if
067                                                  not check:
068 67             check = os.curdir
069 68          self.cwd.set(check)
070 69          self.doLS()
071 70
072 71      def doLS(self, ev=None):
073 72          error = ''
074 73          tdir = self.cwd.get()
075 74          if
076                                                  not tdir: tdir =
os.curdir
077 75
078 76          if
079                                                  not
os.path.exists(tdir):
080 77             error = tdir + ': no such file'
081 78          elif
082                                                  not
os.path.isdir(tdir):
083 79             error = tdir + ': not a directory'
084 80
085 81          if error:
086 82              self.cwd.set(error)
087 83              self.top.update()
088 84              sleep(2)
089 85              if not (hasattr(self, 'last') \
090 86                  and self.last):
091 87                  self.last = os.curdir
092 88              self.cwd.set(self.last)
093 89              self.dirs.config(\
094 90                  selectbackground='LightSkyBlue')
095 91              self.top.update()
096 92              return <$nopage>
097 93
098 94          self.cwd.set(\
099 95              'FETCHING DIRECTORY CONTENTS…')
100 96          self.top.update()
101 97          dirlist = os listdir(tdir)
102 98          dirlist sort()
103 99          os.chdir(tdir)
104 100         self.dirl.config(text=os.getcwd())
105 101         self.dirs.delete(0, END)
106 102         self.dirs.insert(END, os.curdir)
107 103         self.dirs.insert(END, os.pardir)
108 104         for eachFile in dirlist:
109 105             self.dirs.insert(END, eachFile)
110 106         self.cwd.set(os.curdir)
111 107         self.dirs.config(\
112 108             selectbackground='LightSkyBlue')
113 109
114 110     def main():
115 111         d = DirList(os.curdir)
116 112         mainloop()
117 113
```

```
118 114    if __name__ == '__main__':
119 115        main()
120  <$nopage>
```

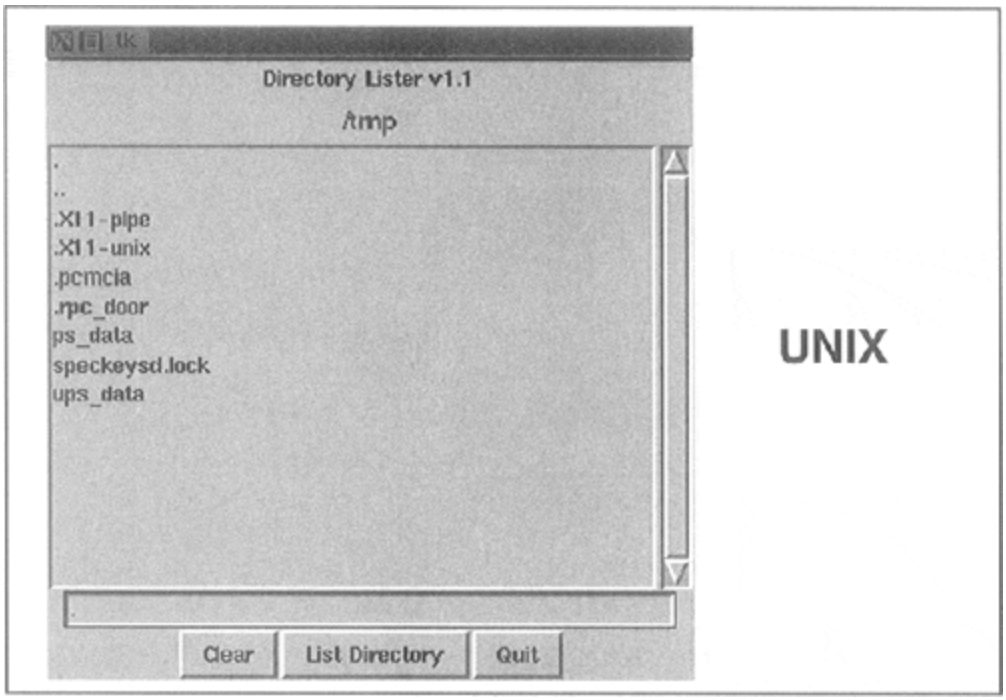
In Figure18-5, we present what this GUI looks like in a Windows environment:

**Figure 18-5. List Directory GUI Application in Windows (`listdir.py`)**




The Unix version of this application is given in Figure18-6.

**Figure 18-6. List Directory GUI Application in UNIX (`listdir.py`)**




### Line-by-line explanation

#### Lines 1–5

These first few lines contain the usual Unix start-up line and importation of the `os` module, the `time.sleep()` function, and all attributes of the `Tkinter` module.

#### Lines 9–13

These lines define the constructor for the `DirList` class, an object which represents our application. The first `Label` we create contains the main title of the application and the version number.

#### Lines 15–19

We declare a Tk variable named `cwd` to hold the name of the directory we are on—we will see where this comes in handy later. Another `Label` is created to display the name of the current directory.

#### Lines 21–30

This section defines the core part of our GUI, the `Listbox dirs,` which contain the list of files of the directory that is being listed. A `Scrollbar` is employed to allow the user to move through a listing if the number of files exceeds the size of the `Listbox.` Both of these widgets are contained in a `Frame` widget. `Listbox` entries have a callback (`setdirandgo`) tied to them using the `Listbox bind()` method.

*Binding* means to tie a keystroke, mouse action, or some other event to a call back to be executed when such an event is generated by the user. `setdirandgo()` will be called if any item in the `Listbox` is doubleclicked. The `Scrollbar` is tied to the `Listbox` by calling the `Scrollbar.config()` method.

**Lines 32–35**

We then create a text `Entry` field for the user to enter the name of the directory he or she wants to traverse to and see its files listed in the `Listbox.` We add a RETURN or Enter key binding to this text entry field so that the user can hit RETURN as an alternative to pressing a button. The same applies for the mouse binding we saw above in the `Listbox.` When the user doubleclicks on a `Listbox` item, it has the same effect as the user's entering the directory name manually into the text `Entry` field and pressing the "go" button.

**Lines 37–54**

We then define a `Button` frame (`bfm`) to hold our three buttons, a "clear" button (`clr`), "go" button (`ls`), and a "quit" button (`quit`). Each button has its own different configuration and callbacks, if pressed.

**Lines 56–58**

The final part of the constructor initializes the GUI program, starting with the current working directory.

**Lines 60–61**

The `clrDir()` method clears the `cwd` Tk string variable, which contains the current directory which is "active." This variable is used to keep track of what directory we are in and, more importantly, helps keep track of the previous directory in case errors arise. You will notice the `ev` variables in the callback functions with a default value of `None.` Any such values would be passed in by the windowing system. They may or may not be used in your callback.

**Lines 63–71**

The `setDirAndGo()` method sets the directory to traverse to and issues the call to the method that makes it all happen, `doLS().`

**Lines 73–108**

`doLS()` is, by far, the key to this entire GUI application. It performs all the safety checks (e.g., is the destination a directory and does it exist?). If there is an error, the last directory is reset to be the current directory. If all goes well, it calls `os.listdir()` to get the actual set of files and replaces the listing in the `Listbox.` While the background work

is going on to pull in the new directory's information, the highlighted blue bar becomes a bright red. When the new directory has been installed, it reverts to blue.

### Lines 110–115

The last pieces of code in `listdir.py` represent the main part of the code. `main()` is executed only if this script is invoked directly, and when `main()` runs, it creates the GUI application, then calls `mainloop()` to start the GUI, which is passed control of the application.

We leave all other aspects of the application as an exercise to the reader, recommending that it is easier to view the entire application as a combination of a set of widgets and functionality. If you see the individual pieces clearly, then the entire script will not appear as daunting.

We hope that we have given you a good introduction to GUI programming with Python and Tkinter. Remember that the best way to get familiar with Tkinter programming is by practicing and stealing a few examples! The Python distribution comes with a large number of demonstration applications (see the `Demo` directory) that you can study. And as we mentioned earlier, there is also an entire text devoted to Tkinter programming.

One final note: do you still doubt the ability of Tkinter to produce a commercial application? Take a close look at IDLE. IDLE itself is a Tkinter application (written by Guido)!

## Related Modules and Other GUIs

There are other GUI development systems which can be used with Python. We present the appropriate modules along with their corresponding window systems in Table 18.2.

**Table 18.2. GUI Systems Available Under Python**

| *GUI Module or System* | *Description* |
|---|---|
| ***Other Tkinter Modules*** | |
| `Pmw` | Python Mega Widgets |
| ***Open Source*** | |
| `wxPython` | wxWindows |
| `PyGTK` | GTK+/GNOME/Glade/GIMP |
| `PyQt/PyKDE` | Qt/KDE |
| ***Commercial*** | |
| `win32ui` | Microsoft MFC |
| `swing` | Sun Microsystems Java/Swing |

## Exercises

*Client-Server Architecture.* Describe the roles of a windows (or windowing) server and a windows client.

*Object-Oriented Programming.* Describe the relationship between child and parent windows.

`Label` *widgets.* Update the `tkhello1.py` script to display your own message instead of "Hello World!"

`Label` *and* `Button` *widgets.* Update the `tkhello3.py` script so that there are three new buttons in addition to the QUIT button. Pressing any of the three buttons will result in changing the text label so that it will then contain the text of the `Button` (widget) that was pressed.

`Label, Button,` *and* `Radiobutton` *widgets.* Modify your solution to the previous problem so that there are three `Radiobuttons` presenting the choices of text for the `Label.` There are two buttons: the QUIT button and an "Update" button. When the Update button is pressed, the text label will then be changed to contain the text of the selected `Radiobutton.` If no `Radiobutton` has been checked, the `Label` will remain unchanged.

`Label, Button,` *and* `Entry` *widgets.* Modify your solution to the previous problem so that the three `Radiobuttons` are replaced by a single `Entry` text field widget with a default value of "Hello World!" (to reflect the initial string in the `Label`). The `Entry` field can be edited by the user with a new text string for the `Label` which will be updated if the Update button is pressed.

`Label` *and* `Entry` *Widgets and Python I/O.* Create a GUI application that provides an `Entry` field where the user can provide the name of a text file. Open the file and read it, displaying its contents in a `Label.` EXTRA CREDIT (Menus): replace the `Entry` widget with menu that has a File Open option that pops up a window to allow the user to specify the file to read. Also add an Exit or Quit option to the menu rather than having a QUIT button.

*Simple Text Editor.* Use your solution to the previous problem to create a simple text editor. A file can be created from scratch or read and displayed into a `Text` widget which can be edited by the user. When the user quits the application (either with the QUIT button or the Quit/Exit menu option), the user is prompted whether to save the changes. EXTRA CREDIT: interface your script to a spellchecker and add a button or menu option to spellcheck the file. The words which are misspelled should be highlighted by using a different foreground or background color in the `Text` widget.

*Multithreaded Chat Applications.* The chat programs from Chapters 13, 16, and 17 need completion. Create a fully-functional multithreaded chat server. A GUI is not really necessary for the server unless you want to create one as a front-end to its configuration, i.e., port number, name, connection to a name server, etc. Create a multithreaded chat client which has separate threads to monitor user input (and sends the message to the server for broadcast) and another thread to accept incoming messages to display to the user. The client front-end GUI should have two portions of the chat window: a larger

section with multiple lines to hold all the dialog, and a smaller text entry field to accept input from the user.

# Chapter 19. Web Programming

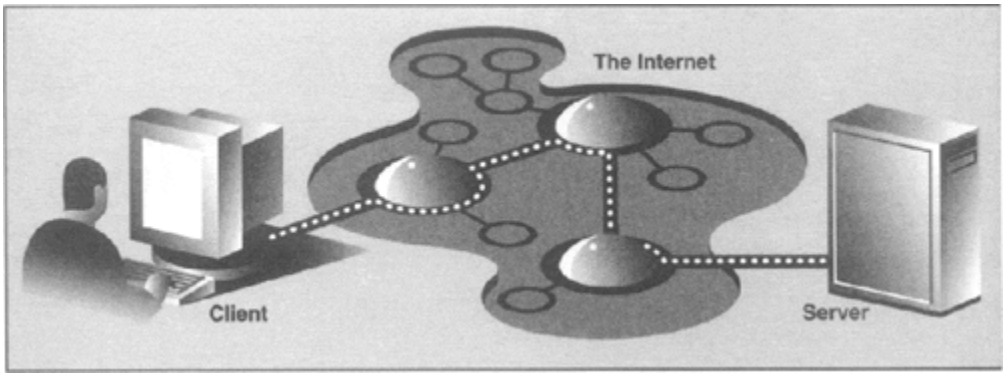
## Introduction

No Python book would be complete without discussing how to do Web programming, one of the main avenues in which people discover Python. In fact, one of the very first Python books was named, "Internet Programming with Python" (unfortunately out-of-print). This introductory chapter on web programming will give you a quick and high-level overview of the kinds of things you can do with Python on the Internet, from Web surfing to creating user feedback forms, from recognizing Uniform Resource Locators to generating dynamic Web page output.

### Web Surfing: Client-Server Computing (Again?!?)

Web surfing falls under the same client-server architecture umbrella that we have seen repeatedly. This time, *Web clients are browsers;* applications allow users to seek documents on the World Wide Web. On the other side are *Web servers,* processes which run on an information provider's host computers. These servers wait for clients and their document requests, process them, and return the requested data. As with most servers in a client-server system, Web servers are designed to run "forever." The Web surfing experience is best illustrated by Figure19-1. Here, a user runs a Web client program such as a browser and makes a connection to a Web server elsewhere on the Internet to obtain their information.

**Figure 19-1. Web Client and Web Server on the Internet. A client sends a request out over the Internet to the server, which then responds with the requested data back to the client.**




Clients may issue a variety of requests to Web servers. Such requests may include obtaining a Web page for viewing or submitting a form with data for processing. The

request is then serviced by the Web server, and the reply comes back to the client in a special format for display purposes.

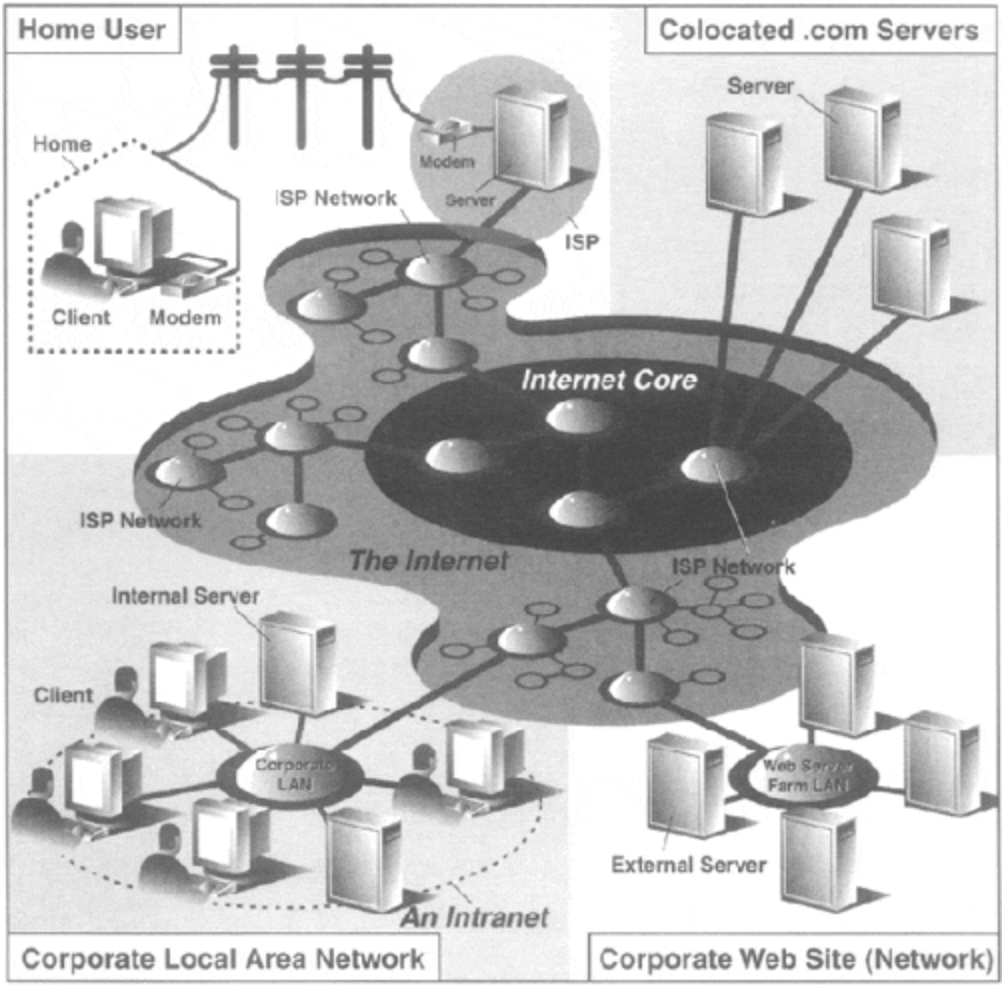
The "language" which is spoken by Web clients and servers, the standard protocol used for Web communication, is called *HTTP,* which stands for *HyperText Transfer Protocol.* HTTP is written "on top of" the TCP and IP protocol suite, meaning that it relies on TCP and IP to carry out its lower-level communication functionality. Its responsibility is not to route or deliver messages—TCP and IP handle that—but to respond to client requests.

HTTP is known as a "stateless" protocol because it does not keep track of information from one client request to the next, similar to the client-server architecture we have seen so far. The server stays running, but client interactions are singular events structured in such a way that once a client request is serviced, it quits. New requests can always be sent, but they are considered separate service requests. Because of the lack of context per request, you may notice that some URLs have a long set of variables and values chained as part of the request to provide some sort of state information. Another alternative is the use of "cookies"—static data stored on the client side which generally contains state information as well. In later parts of this chapter, we will look at how to use both long URLs and cookies to maintain state information.

## The Internet

The Internet is a moving and fluctuating "cloud" or "pond" of interconnected clients and servers scattered around the globe. Communication between client and server consists of a series of connections from one lily pad on the pond to another, with the last step connecting to the server. As a client user, all this detail is kept hidden from your view. The abstraction is to have a direct connection between you the client and the server you are "visiting," but the underlying HTTP, TCP, and IP protocols are hidden underneath, doing all of the dirty work. Information regarding the intermediate "nodes" is of no concern or consequence to the general user anyway, so it's good that the implementation is hidden. Figure19-2 shows an expanded view of the Internet.

**Figure 19-2. A Grand View of the Internet. The left side illustrates where you would find Web clients while the right side hints as to where Web servers are typically located.**




As you can see from the figure, the Internet is made up of multiply-interconnected networks, all working with some sense of (perhaps disjointed) harmony. The left half of the diagram is focused on the Web clients, users who are either at home dialed-in to their *Internet Service Provider* (ISP) or at work on their company's *Local Area Network* (LAN).

The right hand side of the diagram concentrates more on Web servers and where they can be found. Corporations with larger Web sites will typically have an entire "Web server farm" located at their ISP. Such physical placement is called *colocation,* meaning that a company's servers are "co-located" at an ISP along with machines from other corporate customers. These servers are either all providing different data to clients or are part of a redundant system with duplicated information designed for heavy demand (high number of clients). Smaller corporate Web sites may not require as much hardware and networking gear, and hence, may only have one or several colocated servers at their ISP.

In either case, most colocated servers are stored with a larger ISP sitting on a network *backbone,* meaning that they have a "fatter" (meaning wider) and presumably faster connection to the Internet—closer to the "core" of the Internet, if you will. This permits clients to access the servers quickly—being on a backbone means clients do not have to

hop across as many networks to access a server, thus allowing more clients to be serviced within a given time period.

One should also keep in mind that although Web surfing is the most common Internet application, it is not the only one and is certainly not the oldest. The Internet predates the Web by almost three decades. Before the Web, the Internet was mainly used for educational and research purposes. Most of the systems on the Internet run Unix, a multi-user operating system, and many of the original Internet protocols are still around today.

Such protocols include telnet (allows for users to login to a remote host on the Internet and still in use today), FTP (the File Transfer Protocol which enables users to share files and data via uploading or downloading and also still in use today), gopher (the precursor to the Web search engine—a "gopher"-like piece of software that "tunneled the Internet" looking for the data that you were interested in), SMTP or Simple Mail Transfer Protocol (the protocol used for one of the oldest and most widely used Internet applications: electronic mail), and NNTP (News-to-News Transfer Protocol).

Since one of Python's initial strengths was Internet programming, you will find support for all of the protocols discussed above in addition to many others. We differentiate between "Internet programming" and "Web programming" by stating that the latter pertains only to applications developed specifically for Web applications, i.e., Web clients and servers, our focus for this chapter. Internet programming covers a wider range of applications, including some of the Internet protocols we previously mentioned, such as FTP, SMTP, etc., as well as network and socket programming in general, as we discussed in a previous chapter.

## Web Surfing with Python: Creating Simple Web Clients

One thing to keep in mind is that a browser is only one type of Web client. Any application that makes a request for data from a Web server is considered a "client." Yes, it is possible to create other clients which retrieve documents or data off the Internet. One important reason to do this is that a browser provides only limited capacity, i.e., it is used primarily for viewing and interacting with Web sites. A client program, on the other hand, has the ability to do more—it can not only download data, it can also store it, manipulate it, or perhaps even transmit it to another location or application.

Applications which use the `urllib` module to download or access information on the Web [using either `urllib.urlopen()` or `urllib.urlretrieve()`] can be considered a simple Web client. All you need to do is provide a valid Web address.

### Uniform Resource Locators

Simple Web surfing involves using Web addresses called *Uniform Resource Locators* (URLs). Such addresses are used to locate a document on the Web or to call a CGI program to generate a document for your client. URLs are part of a larger set of identifiers known as *URIs* (*Uniform Resource Identifiers*). This superset was created in anticipation of other naming conventions which have yet to be developed. A URL is

simply a URI which uses an existing protocol or scheme (i.e., http, ftp, etc.) as part of its addressing. To complete this picture, we'll add that non-URL URIs are sometimes known as *URNs* (*Uniform Resource Names*), but because URLs are the only URIs in use today, you really don't hear much about URIs or URNs.

Like street addresses, Web addresses have some structure. An American street address usually is of the form "number street designation," i.e., 123 Main Street. It differs from other countries, which have their own rules. A URL is of the format:

```
prot_sch://net_loc/path;params?query#frag
```

Table 19.1 describes each of the components.

**Table 19.1. Web Address Components**

| URL Component | Description |
|---|---|
| `prot_sch` | network protocol or download scheme |
| `net_loc` | location of server (and perhaps user information) |
| `path` | slash ( / ) delimited path to file or CGI application |
| `params` | optional parameters |
| `query` | ampersand ( & ) delimited set of "key=value" pairs |
| `frag` | fragment to a specific anchor within document |

`net_loc` can be broken down into several more components, some required, others optional. The `net_loc` string looks like this:

```
user:passwd@host:port
```

These individual components are described in Table 19.2.

| Table 19.2. Network Location Components | |
|---|---|
| ***net_loc*** | |
| Component | Description |
| *user* | user name or login |
| *passwd* | user password |
| *host* | name or address of machine running Web server [required] |
| *port* | port number (if not 80, the default) |

Of the four, the `host` name is the most important. The `port` number is necessary only if the Web server is running on a different port number from the default. (If you aren't sure what a port number is, go back to Chapter 16.)

User names and perhaps passwords are used only when making FTP connections, and even then, they usually aren't necessary because the majority of such connections are "anonymous."

Python supplies two different modules, each dealing with URLs in completely different functionality and capacities. One is `urlparse`, and the other is `urllib`. We will briefly introduce some of their functions here.

## `urlparse` Module

The `urlparse` module provides basic functionality with which to manipulate URL strings. These functions include `urlparse()`, `urlunparse()`, and `urljoin()`.

### `urlparse.urlparse()`

`urlparse()` breaks up a URL string into some of the major components described above and has the following syntax:

```
urlparse(urlstr, defProtSch=None, allowFrag=None)
```

`urlparse()` parses *`urlstr`* into a 6-tuple (`prot_sch, net_loc, path, params, query, frag`). Each of these components has been described above. *`defProtSch`* indicates a default network protocol or download scheme in case one is not provided in `urlstr`. *`allowFrag`* is a flag that signals whether or not a fragment part of a URL is allowed. Here is what `urlparse()` outputs when given a URL:

```
>>> urlparse.urlparse('http://www.python.org/doc/FAQ.html')
('http',  'www.python.org', '/doc/FAQ.html', '', '', '')
```

### urlparse.urlunparse()

`urlunparse()` does the exact opposite of `urlparse()`—it merges a 6-tuple `(prot_sch, net_loc, path, params, query, frag)`—*`urltup`*, which could be the output of `urlparse()`, into a single URL string and returns it. Accordingly, we state the following equivalence:

```
urlunparse(urlparse(urlstr)) = urlstr
```

You may have already surmised that the syntax of `urlunparse()` is as follows:

```
urlunparse(urltup)
```

### urlparse.urljoin()

The `urljoin()` function is useful in cases where many related URLs are needed, for example, the URLs for a set of pages to be generated for a Web site. The syntax for `urljoin()` is:

```
urljoin(baseurl, newurl, allowFrag=None)
```

`urljoin()` takes *`baseurl`* and joins its base path (*`net_loc`* plus the full path up to, but not including, a file at the end) with *`newurl`*. For example:

```
>>> urlparse.urljoin('http://www.python.org/doc/FAQ.html', \
… 'current/lib/lib.htm')
'http://www.python.org/doc/current/lib/lib.html'
```

A summary of the functions in `urlparse` can be found in Table 19.3

**Table 19.3. Core `urlparse` Module Functions**

| `urlparse` *Functions* | Description |
|---|---|
| `urlparse(urlstr, defProtSch=None, allowFrag=None)` | parses `urlstr` into separate components, using `defProtSch` if the protocol or scheme is not given in `urlstr`; `allowFrag` determines whether a URL fragment is allowed |
| `urlunparse(urltup)` | unparses a tuple of URL data (`urltup`) into a single URL string |
| `urljoin(baseurl, newurl,allowFrag=None)` | merges the base part of the `baseurl` URL with `newurl` to form a complete URL; `allowFrag` is the same as for `urlparse()` |

## `urllib` Module

**NOTE**

Unless you are planning on writing a more lower-level network client, the `urllib` module provides all the functionality you need. `urllib` provides a high-level Web communication library, supporting the basic Web protocols, HTTP, FTP, and Gopher, as well as providing access to local files. Specifically, the functions of the `urllib` module are designed to download data (from the Internet, local network, or local host) using the aforementioned protocols. Use of this module generally obviates the need for using the `httplib`, `ftplib`, and `gopherlib` modules unless you desire their lower-level functionality. In those cases, such modules can be considered as alternatives. (Note: most modules named `*lib` are generally for developing clients of the corresponding protocols. This is not always the case, however, as perhaps `urllib` should then be renamed as "internetlib" or something similar!)

The `urllib` module provides functions to download data from given URLs as well as encoding and decoding strings to make them suitable for including as part of valid URL strings. The functions functions which we will be looking at in this upcoming section

include: `urlopen()`, `urlretrieve()`, `quote()`, `quote_plus()`, `unquote()`, `unquote_plus()`, and `urlencode()`.

We will also look at some of the methods available to the file-like object returned by `urlopen()`.

### `urllib.urlopen()`

`urlopen()` opens a Web connection to the given URL string and returns a file-like object. It has the following syntax:

```
urlopen(urlstr, postQueryData=None)
```

`urlopen()` opens the URL pointed to by *`urlstr`*. If no protocol or download scheme is given, or if a "file" scheme is passed in, `urlopen()` will open a local file.

For all HTTP requests, the normal request type is "GET." In these cases, the query string provided to the Web server (key-value pairs encoded or "quoted," such as the string output of the `urlencode()` function [see below]), should be given as part of *`urlstr`*.

If the "POST" request method is desired, then the query string (again encoded) should be placed in the *`postQueryData`* variable. (For more information regarding the GET and POST request methods, refer to any general documentation or texts on programming CGI applications—which we will also discuss below. GET and POST requests are the two ways to "upload" data to a Web server.

When a successful connection is made, `urlopen()` returns a file-like object as if the destination was a file opened in read mode. If our file object is f, for example, then our "handle" would support the expected read methods such as `f.read()`, `f.readline()`, `f.readlines()`, `f.close()`, and `f.fileno()`.

In addition, a `f.info()` method is available which returns the *MIME* (*Multipurpose Internet Mail Extension*) headers. Such headers give the browser information regarding which application can view returned file types. For example, the browser itself can view HTML (*Hypertext Markup Language*) or plain text type files as well as *GIF* (*Graphics Interchange Format*) and *JPEG* (*Joint Photographic Experts Group*) graphics files. Other files such as multimedia or specific document types require external applications in order to view.

Finally, a `geturl()` method exists to obtain the true URL of the final opened destination, taking into consideration any redirection which may have occurred. A summary of these file-like object methods is given in Table 19.4.

**Table 19.4. `urllib.urlopen()` File-like Object Methods**

| **urlopen()** Object Methods | Description |
|---|---|
| *f*. read ([*bytes*]) | reads all or bytes bytes from f |
| *f*.readline() | reads a single line from f |
| *f*.readlines() | reads a all lines from f into a list |
| *f*.close() | closes URL connection for f |
| *f*.fileno() | returns file number of f |
| *f*.info() | gets MIME headers of f |
| *f*.geturl() | returns true URL opened for f |

### urllib.urlretrieve()

urlretrieve() will do some quick and dirty work for you if you are interested in working with a URL document as a whole. Here is the syntax for urlretrieve():

```
urlretrieve(urlstr, localfile=None, downloadStatusHook=None)
```

Rather than reading from the URL like urlopen() does, urlretrieve() will simply download the entire HTML file located at *urlstr* to your local disk. It will store the downloaded data into *localfile* if given or a temporary file if not. If the file has already been copied from the Internet or if the file is local, no subsequent downloading will occur.

The *downloadStatusHook*, if provided, is a function that is called after each block of data has been downloaded and delivered. It is called with the following three arguments: number of blocks read so far, the block size in bytes, and the total (byte) size of the file. This is very useful if you are implementing "download status" information to the user in a text-based or graphical display.

`urlretrieve()` returns a 2-tuple, (*`filename, mime_hdrs`*). *`filename`* is the name of the local file containing the downloaded data. *`mime_hdrs`* is the set of MIME headers returned by the responding Web server. For more information, see the `Message` class of the `mimetools` module. *`mime_hdrs`* is `None` for local files.

For an example using `urlretrieve()`, take a look at Example 11.2 `(grabweb.py)`.

### `urllib.quote()` and `urllib.quote_plus()`

The `quote*()` functions take URL data and "encodes" them so that they are "fit" for inclusion as part of a URL string. In particular, certain special characters that are unprintable or cannot be part of valid URLs acceptable to a Web server must be converted. This is what the `quote*()` functions do for you. Both `quote*()` functions have the following syntax:

```
quote(urldata, safe='/')
```

Characters that are never converted include commas, underscores, periods and dashes as well as alphanumerics. All others are subject to conversion. In particular, the disallowed characters are changed to their hexadecimal ordinal equivalents prepended with a percent sign ( `%` ), i.e., "`%xx`" where "`xx`" is the hexadecimal representation of a character's ASCII value. When calling `quote*()`, the *`urldata`* string is converted to an equivalent string that can be part of a URL string. The *`safe`* string should contain a set of characters which should also *not* be converted. The default is the slash ( `/` ).

`quote_plus()` is similar to `quote()` except that it also encodes spaces to plus signs ( `+` ). Here is an example using `quote()` vs. `quote_plus()`:

```
>>> name = 'joe mama'
>>> number = 6
>>> base = 'http://www/~foo/cgi-bin/s.py'
>>> final = '%s?name=%s&num=%d' % (base, name, number)
>>> final
'http://www/~foo/cgi-bin/s.py?name=joe mama&num=6'
>>>
>>> urllib.quote(final)
'http:%3a//www/%7efoo/cgi-bin/s.py%3fname%3djoe%20mama%26num%3d6'
>>>
>>> urllib.quote_plus(final)
'http%3a//www/%7efoo/cgi-bin/s.py%3fname%3djoe+mama%26num%3d6'
```

### `urllib.unquote()` and `urllib.unquote_plus()`

As you have probably guessed, the `unquote*()` functions do the exact opposite of the `quote*()` functions—they convert all characters encoded in the "`%xx`" fashion to their ASCII equivalents. The syntax of `unquote*()` is as follows:

```
unquote*(urldata)
```

Calling `unquote()` will decode all URL-encoded characters in `urldata` and return the resulting string. `unquote_plus()` will also convert plus signs back to space characters.

### urllib.urlencode()

`urlencode()`, recently added to Python (as of version 1.5.2) takes a dictionary of key-value pairs and encodes them to be included as part of a query in a CGI request URL string. The pairs are in "`key=value`" format and are delimited by ampersands ( `&` ). Furthermore, the keys and their values are sent to `quote_plus()` for proper encoding. Here is an example output from `urlencode()`:

```
>>> aDict = { 'name': 'Georgina Garcia', 'hmdir': '~ggarcia' }
>>> urllib.urlencode(aDict)
'name=Georgina+Garcia&hmdir=%7eggarcia'
```

There are other functions in `urllib` and `urlparse` which we did not have the opportunity to cover here. Refer to the documentation for more information.

### Secure Socket Layer support

The `urllib` module has been modified for Python 1.6 so that it now supports opening HTTP connections using the Secure Socket Layer (SSL). The core change to add SSL is implemented in the `socket` module. Consequently, the `urllib` and `httplib` modules were updated to support URLs using the "https" connection scheme. Note, however, that as of time of publication, only HTTP requests using SSL have been implemented. The future may see additional updates to the other protocols supported by the `urllib` module, such as FTP.

A summary of the `urllib` functions discussed in this section can be found in Table 19.5.

**Table 19.5. Core `urllib` Module Functions**

| `urllib` Functions | Description |
|---|---|
| `urlopen(urlstr,` | opens the URL `urlstr,` sending the query data in |

| | |
|---|---|
| *postQueryData*=None) | *postQueryData* if a POST request |
| urlretrieve(*urlstr*, *localfileV*=None, *downloadStatusHook*=None) | downloads the file located at the *urlstr* URL to *localfile* or a temporary file if *localfile* not given; if present, *downloaStatusHook* is a function which can receive download statistics |
| quote(*urldata*, *safe*='/') | encodes invalid URL characters of *urldata*; characters in *safe* string are also not encoded |
| quote_plus(*urldata*, *safe*='/') | same as quote() except encodes spaces as plus signs |
| unquote(*urldata*) | decodes encoded characters of *urldata* |
| unquote_plus(*urldata*) | same as unquote() but converts plus signs to spaces |
| urlencode(*dict*) | encodes the key-value pairs of *dict* into a valid string for CGI queries and encodes the key and value strings with quote_plus() |

## Advanced Web Clients

Web browsers are basic Web clients. They are used primarily for searching and downloading documents from the Web. Advanced clients of the Web are those applications which do more than download single documents from the Internet.

One example of an advanced Web client is a *crawler* (a.k.a. *spider, robot*). These are programs which explore and download pages from the Internet for different reasons, some of which include:

Indexing or cataloging into a large search engine such as Google, Alta Vista, or Yahoo!,

Offline browsing—downloading documents onto a local hard disk and rearranging hyperlinks to create almost a mirror image for local browsing,

Downloading and storing for historical or archival purposes, or

Web page caching to save superfluous downloading time on Web site revisits.

The crawler we present below, `crawl.py`, takes a starting Web address (URL), downloads that page and all other pages whose links appear in succeeding pages, but only those which are in the same domain as the starting page. Without such limitations, you will run out of disk space! The source for `crawl.py` follows:

**Example 19.1. An Advanced Web Client: a Web Crawler (`crawl.py`)**

The crawler consists of two classes, one to manage the entire crawling process (`Crawler`), and one to retrieve and parse each downloaded Web page (`Retriever`).

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   from sys import argv
004 4   from os import makedirs, unlink
005 5   from os.path import dirname, exists, isdir, splitext
006 6   from string import replace, find, lower
007 7   from htmllib import HTMLParser
008 8   from urllib import urlretrieve
009 9   from urlparse import urlparse, urljoin
010 10  from formatter import DumbWriter, AbstractFormatter
011 11  from cStringIO import StringIO
012 12
013 13  class Retriever:   #   download Web pages
014 14
015 15      def __init__(self, url):
016 16          self.url = url
017 17          self.file = self.filename(url)
018 18
019 19      def filename(self, url, deffile='index.htm'):
020 20          parsedurl = urlparse(url, 'http:', 0)# parse path
021 21          path = parsedurl[1] + parsedurl[2]
022 22          text = splitext(path)
023 23          if ext[1] == '':   # no file, use default
024 24              if newpath[-1] == '/':
025 25                  path = path + deffile
026 26          else: <$nopage>
027 27                  path = path + '/' + deffile
028 28          dir = dirname(path)
029 29          if not isdir(dir):    # create archive dir if nec.
030 30              if exists(dir): unlink(dir)
031 31              makedirs(dir)
032 32          return path
033 33
034 34      def download(self):  # download Web page
035 35          try: <$nopage>
036 36              retval = urlretrieve(self.url, self.file)
037 37          except IOError:
038 38              retval = ('*** ERROR: invalid URL "%s"' %\
039 39                                          self.url,)
040 40          return retval
041 41
042 42      def parseAndGetLinks(self):    # parse HTML, save links
043 43          self.parser = HTMLParser(AbstractFormatter(\
044 44                          DumbWriter(StringIO())))
```

```
045 45          self.parser.feed(open(self.file).read())
046 46          self.parser.close()
047 47          return self.parser.anchorlist
048 48
049 49  class Crawler:     # manage entire crawling process
050 50
051 51      count = 0       # static downloaded page counter
052 52
053 53      def __init__(self, url):
054 54          self.q = [url]
055 55          self.seen = []
056 56          self.dom = urlparse(url)[1]
057 57
058 58      def getPage(self, url):
059 59          r = Retriever(url)
060 60          retval = r.download()
061 61          if retval[0] == '*':  # error situation, do not parse
062 62              print retval, '... skipping parse'
063 63              return <$nopage>
064 64          Crawler.count = Crawler.count + 1
065 65          print '\n(', Crawler.count, ')'
066 66          print 'URL:', url
067 67          print 'FILE:', retval[0]
068 68          self.seen.append(url)
069 69
070 70          links = r.parseAndGetLinks() # get and process links
071 71          for eachLink in links:
072 72              if eachLink[:4] != 'http' and \
073 73                      find(eachLink, '://') == -1:
074 74                  eachLink = urljoin(url, eachLink)
075 75              print '* ', eachLink,
076 76
077 77              if find(lower(eachLink), 'mailto:') != -1:
078 78                  print '... discarded, mailto link'
079 79                  continue <$nopage>
080 80
081 81              if eachLink not in self.seen:
082 82                  if find(eachLink, self.dom) == -1:
083 83                      print '... discarded, not in domain'
084 84                  else: <$nopage>
085 85                      if eachLink not in self.q:
086 86                          self.q.append(eachLink)
087 87                          print '... new, added to Q'
088 88                      else: <$nopage>
089 89                  print '... discarded, already in Q'
090 90              else: <$nopage>
091 91                  print '... discarded, already processed'
092 92
093 93  def go(self):# process links in queue
094 94          while self.q:
095 95              url = self.q.pop()
096 96              self.getPage(url)
097 97
098 98  def main():
099 99      if len(argv) > 1:
100 100         url = argv[1]
101 101     else: <$nopage>
```

```
102 102         try: <$nopage>
103 103             url = raw_input('Enter starting URL: ')
104 104         except (KeyboardInterrupt, EOFError):
105 105             url = ''
106 106
107 107     if not url: return <$nopage>
108 108     robot = Crawler(url)
109 109     robot.go()
110 110
111 111 if __name__ == '__main__':
112 112     main()
113  <$nopage>
```

**Line-by-line (Class-by-class) explanation:**

**Lines 1– 11**

The top part of the script consists of the standard Python Unix start-up line and the importation of various module attributes which are employed in this application.

**Lines 13 – 47**

The `Retriever` class has the responsibility of downloading pages from the Web and parsing the links located within each document, adding them to the "to-do" queue if necessary. A `Retriever` instance object is created for each page which is downloaded from the net. `Retriever` consists of several methods to aid in its functionality: a constructor (`__init__()`), `filename(), download(),` and `parseAndGetLinks().`

The `filename()` method takes the given URL and comes up with a safe and sane corresponding file name to store locally. Basically, it removes the "`http://`" prefix from the URL and uses the remaining part as the file name, creating any directory paths necessary. URLs without trailing file names will be given a default file name of "`index.htm.`" (This name can be overridden in the call to `filename()`).

The constructor instantiates a `Retriever` object and stores both the URL string and the corresponding file name returned by `filename()` as local attributes.

The `download()` method, as you may imagine, actually goes out to the net to download the page with the given link. It calls `urllib.urlretrieve()` with the URL and saves it to the filename (the one returned by `filename()`). If the download was successful, the `parse()` method is called to parse the page just copied from the network, otherwise an error string is returned.

If the `Crawler` determines that no error has occurred, it will invoke the `parseAndGetLinks()` method to parse newly-downloaded page and determine the cause of action for each link located on that page.

**Lines 49 – 96**

The `Crawler` class is the "star" of the show, managing the entire crawling process, thus only one instance is created for each invocation of our script. The `Crawler` consists of three items stored by the constructor during the instantiation phase, the first of which is `q`, a queue of links to download. Such a list will fluctuate during execution, shrinking as each page is processed and grown as new links are discovered within each downloaded page.

The other two data values for the `Crawler` include `seen`, a list of all the links which "we have seen" (downloaded) already. And finally, we store the domain name for the main link, `dom`, and use that value to determine whether any succeeding links are part of the same domain.

`Crawler` also has of a static data item named `count`. The purpose of this counter is just to keep track of the number of objects we have downloaded from the net. It is incremented for every page successfully download.

`Crawler` has a pair of other methods in addition to its constructor, `getPage()` and `go()`. `go()` is simply the method that is used to start the `Crawler` and is called from the main body of code. `go()` consists of a loop that will continue to execute as long as there are new links in the queue which need to be downloaded. The workhorse of this class though, is the `getPage()` method.

`getPage()` instantiates a `Retriever` object with the first link and lets it go off to the races. If the page was downloaded successfully, the counter is incremented and the link added to the "already seen" list. It looks recursively at all the links featured inside each downloaded page and determine whether any more links should be added to the queue. The main loop in `go()` will continue to process links until the queue is empty, at which time victory is declared.

Links which are: part of another domain, have already been downloaded, are already in the queue waiting to be processed, or are "`mailto:`" links are ignored and not added to the queue.

### Lines 98 – 112

`main()` is executed if this script is invoked directly and is the starting point of execution. Other modules which import `crawl.py` will need to invoke `main()` to begin processing. `main()` needs a URL to begin processing, If one is given on the command-line (for example which this script is invoked directly), it will just go with the one given. Otherwise, the script enters interactive mode prompting the user for a starting URL. With a starting link in hand, the `Crawler` is instantiated and away we go.

One sample invocation of `crawl.py` may look like:

```
% crawl.py
Enter starting URL: http://www.null.com/home/index.html
```

```
( 1 )
URL: http://www.null.com/home/index.html
FILE: www.null.com/home/index.html
* http://www.null.com/home/overview.html … new, added to Q
* http://www.null.com/home/synopsis.html … new, added to Q
* http://www.null.com/home/order.html … new, added to Q
* mailto:postmaster@null.com … discarded, mailto link
* http://www.null.com/home/overview.html … discarded, already in Q
* http://www.null.com/home/synopsis.html … discarded, already in Q
* http://www.null.com/home/order.html … discarded, already in Q
* mailto:postmaster@null.com … discarded, mailto link
* http://bogus.com/index.html … discarded, not in domain

( 2 )
URL: http://www.null.com/home/order.html
FILE: www.null.com/home/order.html
* mailto:postmaster@null.com … discarded, mailto link
* http://www.null.com/home/index.html … discarded, already processed
* http://www.null.com/home/synopsis.html … discarded, already in Q
* http://www.null.com/home/overview.html … discarded, already in Q

( 3 )
URL: http://www.null.com/home/synopsis.html
FILE: www.null.com/home/synopsis.html
* http://www.null.com/home/index.html … discarded, already processed
* http://www.null.com/home/order.html … discarded, already processed
* http://www.null.com/home/overview.html … discarded, already in Q

( 4 )
URL: http://www.null.com/home/overview.html
FILE: www.null.com/home/overview.html
* http://www.null.com/home/synopsis.html … discarded, already processed
* http://www.null.com/home/index.html … discarded, already processed
* http://www.null.com/home/synopsis.html … discarded, already processed
* http://www.null.com/home/order.html … discarded, already processed
```

After execution, a http://www.null.com directory would be created in the local file system, with a `home` subdirectory. Within `home`, all the HTML files processed will be found there.

# CGI: Helping Web Servers Process Client Data

## Introduction to CGI

The Web was initially developed to be a global online repository or archive of (mostly educational and research-oriented) documents. Such pieces of information generally come in the form of static text and usually in HTML (*HyperText Markup Language*). [Many documents also exist in plain text, Adobe *Portable Document Format* (PDF), or *Extensible Markup Language* (XML) format, a generalized markup language.]

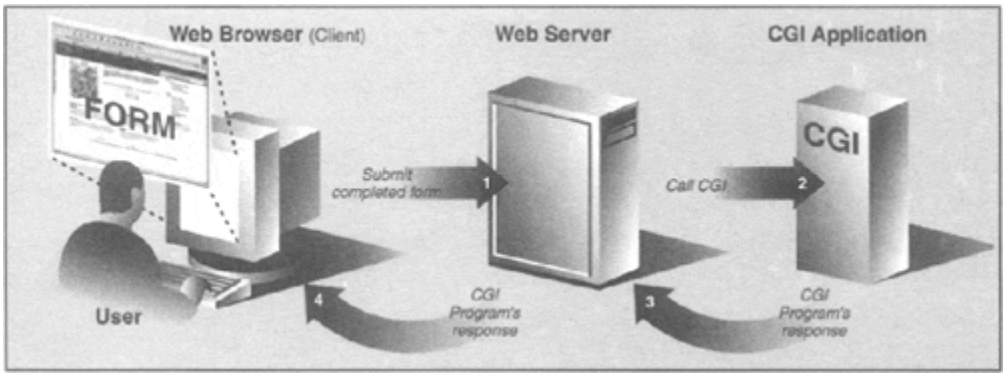
HTML is not as much of a *language* as it is a text formatter, indicating changes in font types, sizes, and styles. The main feature of HTML is in its hypertext capability, document text that is in one way or another highlighted to point to another document in a related context to the original. Such a document can be accessed by a mouse click or other user selection mechanism. These (static) HTML documents live on the Web server and are sent to clients when and if requested.

As the Internet and Web services evolved, there grew a need to process user input. Online retailers needed to be able to take individual orders, and online banks and search engine portals needed to create accounts for individual users. Thus fill-out forms were invented, and became the only way a Web site can get specific information from users (until Java applets came along).This, in turn, required the HTML now be generated on the fly, for each client submitting user-specific data.

Now Web servers are only really good at one thing, getting a user request for a file and returning that file (i.e., an HTML file) to the client. They do not have the "brains" to be able to deal with user-specific data such as those which come from fields. Not being their responsibility, Web servers farm out such requests to external applications which create the dynamically-generated HTML that is returned to the client.

The entire process begins when the Web server receives a client request (i.e., GET or POST) and calls the appropriate application. It then waits for the resulting HTML—meanwhile, the client also waits. Once the application has completed, it passes the dynamically-generated HTML back to the server, who then (finally) forwards it back to the user. This process of the server receiving a form, contacting an external application, receiving and returning the newly-generated HTML takes place through what is called the Web server's *Common Gateway Interface* (CGI). An overview of how CGI works is illustrated in Figure19-3, which shows you the execution and data flow, step-by-step from when a user submits a form until the resulting Web page is returned.

**Figure 19-3. Overview of how CGI Works. CGI represents the interaction between a web server and the application which is required to process a user's form and generate the dynamic HTML that is eventually returned.**




Forms input from the client sent to a Web server may include processing and perhaps some form of storage in a backend database. Just keep in mind that any time there are any

user-filled fields and/or a Submit button or image, it most likely involves some sort of CGI activity.

CGI applications which create the HTML are usually written in one of many higher-level programming languages which have the ability to accept user data, process it, and return value HTML back to the server. Today, these include: Perl, C, C++, or Python, to name a few. In this next section, we will look at how to create CGI applications in Python, with the help of the `cgi` module.

## CGI Applications

A CGI application is slightly different from a typical program. The primary differences are in the input, output, and user interaction aspects of a computer program.

When a CGI script starts, it will have the additional functionality of retrieving the user-supplied data, the input for the program comes from the data via the Web client, not a user on the server machine nor a disk file.

The output differs in that any data sent to standard output will be sent back to the connected Web client rather than to the screen, GUI window, or disk file. The data that is sent back must be a set of valid headers followed by HTML. If it is not and the Web client is a browser, an error (specifically, an Internal Server Error) will occur because Web clients such as browsers understand only valid HTTP data (i.e., MIME headers and HTML).

Finally, as you can probably guess, there is no user interaction with the script. All communication occurs among the Web client (on behalf of a user), the Web server, and the CGI application.

## `cgi` Module

There is one primary class in the `cgi` module which does all the work: the `FieldStorage` class. This class should be instantiated when a Python CGI script begins, as it will read in all the pertinent user information from the Web client (via the Web server). Once this object has been instantiated, it will consist of a dictionary-like object which has a set of key-value pairs. The keys are the names of the form items that were passed in through the form while the values contain the corresponding data.

These values themselves can be one of three objects. They can be `FieldStorage` objects (instances) as well as instances of a similar class called `MiniFieldStorage`, which is used in cases where no file uploads or multiple part form data is involved. `MiniFieldStorage` instances contain only the key-value pair of the name and the data. Lastly, they can be a list of such objects. This occurs when a form contains more than one input item with the same field name.

For simple Web forms, you will usually find all `MiniFieldStorage` instances. All of our examples below pertain only to this general case.

# Building CGI Application

## Generating the Results Page

In Example 19.2, we present the code for a simple Web form, `friends.htm`.

### Example 19.2. Static Form Web Page (`friends.htm`)

This HTML file presents a form to the user with an empty field for the user's name and a set of radio buttons for the user to choose from.

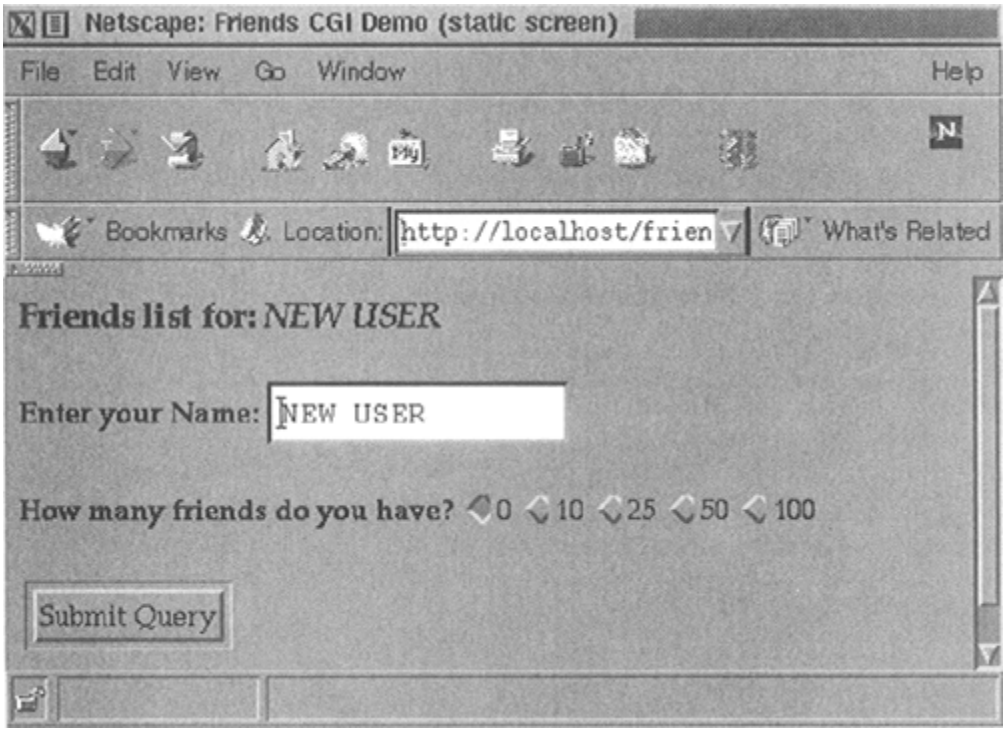
```
 <$nopage>
001 1   <HTML><HEAD><TITLE>
002 2   Friends CGI Demo (static screen)
003 3   </TITLE></HEAD>
004 4   <BODY><H3>Friends list for: <I>NEW USER</I></H3>
005 5   <FORM ACTION="/cgi-bin/friends1.py">
006 6   <B>Enter your Name:</B>
007 7   <INPUT TYPE=text NAME=person SIZE=15>
008 8   <P><B>How many friends do you have?</B>
009 9   <INPUT TYPE=radio NAME=howmany VALUE="0"> CHECKED> 0
010 10  <INPUT TYPE=radio NAME=howmany VALUE="10"> 10
011 11  <INPUT TYPE=radio NAME=howmany VALUE="25"> 25
012 12  <INPUT TYPE=radio NAME=howmany VALUE="50"> 50
013 13  <INPUT TYPE=radio NAME=howmany VALUE="100"> 100
014 14  <P><INPUT TYPE=submit></FORM></BODY></HTML>
015  <$nopage>
```

As you can see in the code, the form contains two input variables: person and howmany. The values of these two fields will be passed to our CGI script, `friends1.py`.

You will notice in our example that we install our CGI script into the default `cgi-bin` directory (see the "Action" link) on the local host. (If this information does not correspond with your development environment, update the form action before attempting to test the Web page and CGI script.) Also, because a METHOD subtag is missing from the form action, all requests will be of the default type, GET. We choose the GET method because we do not have very many form fields, and also, we want our query string to show up in the "Location" (a.k.a. "Address," "Go To") bar so that you can see what URL is sent to the server.

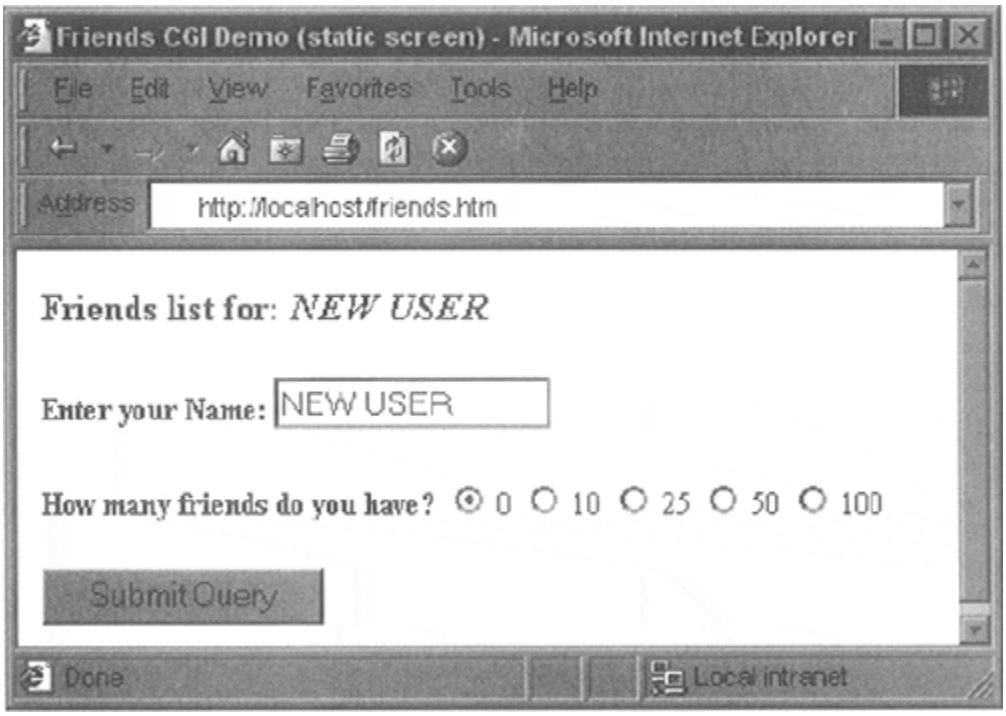
Let's take a look at the screen which is rendered by `friends.htm` in a Web browser. Figure19-4 illustrates what the page would look like using Netscape Communicator 4 in a UNIX environment, while Figure19-5 is an example of using Microsoft IE5 on Windows.

**Figure 19-4. Friends Form Page in Netscape4 on Unix (`friends.htm`)**




The input is entered by the user and the "Submit" button is pressed. (Alternatively, the user can also press the RETURN or Enter key within the text field to cause a similar effect.) When this occurs, the script in Example 19.3, `friends1.py`, is executed via CGI.

**Figure 19-5. Friends Form Page in IE5 on Windows (`friends.htm`)**



### Example 19.3. Results Screen CGI code (friends1.py)

This CGI script grabs the person and howmany fields from the form and uses that data to create the dynamically-generated results screen.

```
 <$nopage>
001   1   #!/usr/bin/env python
002 2
003 3   import cgi
004 4
005 5   reshtml = '''Content-Type: text/html\n
006 6   <HTML><HEAD><TITLE>
007 7   Friends CGI Demo (dynamic screen)
008 8   </TITLE></HEAD>
009 9   <BODY><H3>Friends list for: <I>%s</I></H3>
010 10  Your name is: <B>%s</B><P>
011 11  You have <B>%s</B> friends.
012 12  </BODY></HTML>'''
013 13
014 14  form = cgi.FieldStorage()
015 15  who = form['person'].value
016 16  howmany = form['howmany'].value
017 17  print reshtml % (who, who, howmany)
018  <$nopage>
```

This script contains all the programming power to read the form input and process it, as well as return the resulting HTML page back to the user. All the "real" work in this script takes place in only four lines of Python code (lines 14–17).

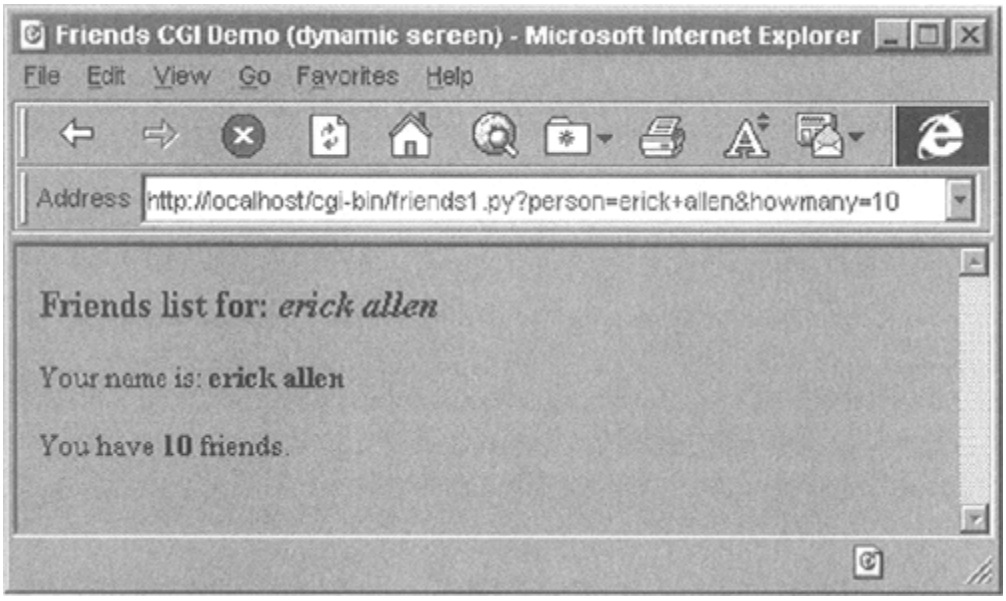
The form variable is our FieldStorage instance, containing the values of the person and howmany fields. We read these into the Python whoand howmany variables, respectively. The reshtml variable contains the general body of HTML text to return, with a few fields filled in dynamically, the data just read in from the form.

**NOTE**

One thing which always nails CGI beginners is that when sending results back to a CGI script, it must return the appropriate HTTP headers first before any HTML. Furthermore, to distinguish between these headers and the resulting HTML, several newline characters must be inserted between both sets of data, as in line 5 of our friends1.py example as well as for the code in the remaining part of the chapter.

One possible resulting screen appears in Figure19-6, assuming the user typed in "erick allen" as the name and clicked on the "10 friends" radio button.

**Figure 19-6. Friends Results Page in IE3 on Windows**




The screen snapshot this time is represented by the older IE3 browser in a Windows environment.

If you are a Web site producer, you may be thinking, "Gee, wouldn't it be nice if I could automatically capitalize this person's name, especially if they forgot?" This can easily be accomplished using Python CGI. (And we shall do so soon!)

Notice how on a GET request that our form variables and their values are added to the form action URL in the "Address" bar. Also, did you observe that the title for the `friends.htm` page has the word "static" in it while the output screen from friends.py has the work "dynamic" in *its* title? We did that for a reason: to indicate that `friends.htm` file is a static text file while the results page is dynamically-generated. In other words, the HTML for the results page did not exist on disk as a text file; rather, it was generated by our CGI script and returned it as if it *was* a local file.

In our next example, we will bypass static files altogether by updating our CGI script to be somewhat more multifaceted.

## Generating Form and Results Pages

We obsolete `friends.html` and merge it into `friends2.py`. The script will now generate both the form page as well as the results page. But how can we tell which page to generate? Well, if there is form data being sent to us, that means that we should be creating a results page. If we do not get any information at all, that tells us that we should generate a form page for the user to enter his or her data.

Our new `friends2.py` script is shown in Example 19.4.

**Example 19.4. Generating Form and Results Pages (`friends2.py`)**

Both `friends.html` and `friends1.py` are merged together as `friends2.py`. The resulting script can now output both form and results pages as dynamically-generated HTML and has the smarts to know which page to output.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import cgi
004 4
005 5  header =  'Content-Type: text/html\n\n'
006 6
007 7  formhtml =  '''<HTML> <HEAD><TITLE>
008 8  Friends CGI Demo</TITLE></HEAD>
009 9  <BODY><H3>Friends list for: <I>NEW USER</I></H3>
010 10 <FORM ACTION="/cgi-bin/friends2.py">
011 11 <B>Enter your Name:</B>
012 12 <INPUT TYPE=hidden NAME=action VALUE=edit>
013 13 <INPUT TYPE=text NAME=person SIZE=15>
014 14 <P><B>How many friends do you have?</B>
015 15 %s
016 16 <P><INPUT TYPE=submit></FORM></BODY></HTML>'''
017 17
018 18 fradio = '<INPUT TYPE=radio NAME=howmany VALUE="%s" %s> %s\n'
019 19
020 20 def showForm():
021 21     friends = ''
022 22     for i in [0, 10, 25, 50, 100]:
023 23         checked = ''
024 24         if i == 0:
025 25            checked = 'CHECKED'
026 26         friends = friends + fradio % \
027 27             (str(i), checked, str(i))
028 28
029 29     print header + formhtml % (friends)
030 30
031 31 reshtml = '''<HTML><HEAD><TITLE>
032 32 Friends CGI Demo</TITLE></HEAD>
033 33 <BODY><H3>Friends list for: <I>%s</I></H3>
034 34 Your name is: <B>%s</B><P>
035 35 You have <B>%s</B> friends.
036 36 </BODY></HTML>'''
037 37
038 38 def doResults(who, howmany):
039 39     print header + reshtml % (who, who, howmany)
040 40
041 41 def process():
042 42     form = cgi.FieldStorage()
043 43     if form.has_key('person'):
044 44         who = form['person'].value
045 45     else: <$nopage>
046 46         who = 'NEW USER'
047 47
048 48     if form.has_key('howmany'):
049 49         howmany = form['howmany'].value
050 50     else: <$nopage>
051 51         howmany = 0
```

```
052 52
053 53      if form.has_key('action'):
054 54          doResults(who, howmany)
055 55      else: <$nopage>
056 56          showForm()
057 57
058 58 if __name__ == '__main__':
059 59      process()
060  <$nopage>
```

So what did we change in our script? Let's take a look at some of the blocks of code in this script.

### Line-by-line explanation

### Lines 1 – 5

In addition to the usual start-up and module import lines, we separate the HTTP MIME header from the rest of the HTML body because we will use it for both types of pages (form page and results page) returned and don't want to duplicate the text. We will add this header string to the corresponding HTML body when it comes time for output to occur.

### Lines 7 – 29

All of this code is related to the now-integrated `friends.htm` form page in our CGI script. We have a variable for the form page text, `formhtml,` and we also have a string to build the list of radio buttons, `fradio.` We could have duplicated this radio button HTML text as it is in `friends.htm,` but we wanted to show how we could use Python to generate more dynamic output—see the **`for`**-loop on lines 22–27.

The `showForm()` function has the responsibility of generating a form for user input. It builds a set of text for the radio buttons, merges those lines of HTML into the main body of `formhtml,` prepends the header to the form, and then returns the entire wad of data back to the client by sending the entire string to standard output.

There are a couple of interesting things to note about this code. The first is the "hidden" variable in the form called `action,` containing the value, "edit" on line 12. This field is the only way we can tell which screen to display (i.e., the form page or the results page). We will see this field come into play in lines 53–56.

Also, observe that we set the 0 radio button as the default by "checking" it within the loop that generates all the buttons. This will also allow us to update the layout of the radio buttons and/or their values on a single line of code (line 18) rather than over multiple lines of text. It will also offer some more flexibility in letting the logic determine which radio button is checked—see the next update to our script, `friends3.py` coming up.

Now you may be thinking, "Why do we need an `action` variable when I could just as well be checking for the presence of `person` or `howmany?`" That is a valid question because yes, you could have just used `person` or `howmany` in this situation.

However, the `action` variable is a more conspicuous presence, in as far as its name as well as what it does—the code is easier to understand. The `person` and `howmany` variables are used for their values while the `action` variable is used as a flag.

The other reason for creating `action` is that we will be using it again to help us determine which page to generate. In particular, we will need to display a form *with* the resence of a `person` variable (rather than a results page)—this will break your code if you are solely relying on there being a `person` variable.

**Lines 31 – 39**

The code to display the results page is practically identical to that of `friends1.py.`

**Lines 41 – 56**

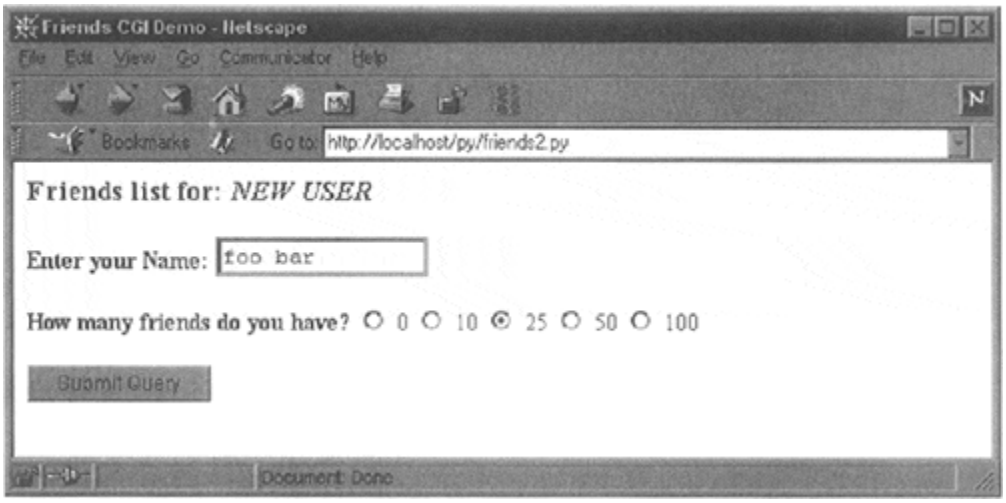
Since there are different pages which can result from this one script, we created an overall `process()` function to get the form data and decide which action to take. The main portion of `process()` will also look familiar to the main body of code in `friends1.py.` There are two major differences, however.

Since the script may or may not be getting the expected fields (invoking the script the first time to generate a form page, for example, will not pass any fields to the server), we need to "bracket" our retrieval of the form fields with **`if`** statements to check if they are even there. Also, we mentioned the `action` field above which helps us decide which page to bring up. The code that performs this determination is in lines 53–56.

In Figure19-8 and Figure19-7, you will see first the form screen generated by our script (with a name entered and radio button chosen), followed by the results page, also generated by our script.

**Figure 19-8. Friends Form Page in Netscape4 on Windows**


**Figure 19-7. Friends Results Page in Netscape4 on Windows**




If you look at the location or "Go to" bar, you will not see a URL referring to a static `friends.htm` file as you did in Figure19-4 or Figure19-5 earlier.

## Fully Interactive Web Sites

Our final example will complete the circle. As in the past, a user enters his or her information from the form page. We then process the data and output a results page. Now we will add a link to the results page that will allow the user to go *back* to the form page, but rather than presenting a blank form, we will fill in the data that the user has already provided. We will also add some error processing to give you an example of how it can be accomplished.

We now present our final update, `friends3.py` in Example 19.5.

**Example 19.5. Full User Interaction and Error Processing (`friends3.py`)**

By adding a link to return to the form page with information already provided, we have come "full circle," giving the user a fully-interactive Web surfing experience. Our application also now performs simple error checking which notifies the user if no radio button was selected.

```
 <$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import cgi
004 4  from urllib import quote_plus
005 5  from string import capwords
006 6
007 7  header = 'Content-Type: text/html\n\n'
008 8  url = '/cgi-bin/friends3.py'
009 9
010 10 errhtml = '''<HTML><HEAD><TITLE>
011 11 Friends CGI Demo</TITLE></HEAD>
012 12 <BODY><H3>ERROR</H3>
013 13 <B>%s</B><P>
014 14 <FORM><INPUT TYPE=button VALUE=Back
015 15 ONCLICK="window.history.back()"></FORM <$nopage>
016 16 </BODY></HTML>'''
017 17
018 18 def showError(error_str):
019 19     print header + errhtml % (error_str)
020 20
021 21 formhtml = '''<HTML><HEAD><TITLE>
022 22 Friends CGI Demo</TITLE></HEAD>
023 23 <BODY><H3>Friends list for: <I>%s</I></H3>
024 24 <FORM ACTION="%s">
025 25 <B>Your Name:</B>
026 26 <INPUT TYPE=hidden NAME=action VALUE=edit>
027 27 <INPUT TYPE=text NAME=person VALUE="%s" SIZE=15>
028 28 <P><B>How many friends do you have?</B>
029 29 %s
030 30 <P><INPUT TYPE=submit></FORM></BODY></HTML>'''
031 31
032 32 fradio = '<INPUT TYPE=radio NAME=howmany VALUE="%s" %s> %s\n'
033 33
034 34 def showForm(who, howmany):
035 35     friends = ''
036 36     for i in [0, 10, 25, 50, 100]:
037 37        checked = ''
038 38        if str(i) == howmany:
039 39           checked = 'CHECKED'
040 40        friends = friends + fradio % \
041 41           (str(i), checked, str(i))
042 42     print header + formhtml % (who, url, who, friends)
043 43
044 44 reshtml = '''<HTML><HEAD><TITLE>
045 45 Friends CGI Demo</TITLE></HEAD>
046 46 <BODY><H3>Friends list for: <I>%s</I></H3>
047 47 Your name is: <B>%s</B><P>
048 48 You have <B>%s</B> friends.
049 49 <P>Click <A HREF="%s">here</A> to edit your data again.
050 50 </BODY></HTML>'''
```

```
051 51
052 52 def doResults(who, howmany):
053 53      newurl = url + '?action=reedit&person=%s&howmany=%s'%\
054 54         (quote_plus(who), howmany)
055 55      print header + reshtml % (who, who, howmany, newurl)
056 56
057 57 def process():
058 58      error = ''
059 59      form = cgi.FieldStorage()
060 60
061 61      if form.has_key('person'):
062 62        who = capwords(form['person'].value)
063 63      else: <$nopage>
064 64        who = 'NEW USER'
065 65
066 66      if form.has_key('howmany'):
067 67        howmany = form['howmany'].value
068 68      else: <$nopage>
069 69        if form.has_key('action') and \
070 70               form['action'].value == 'edit':
071 71           error = 'Please select number of friends.'
072 72        else: <$nopage>
073 73          howmany = 0
074 74
075 75      if not error:
076 76         if form.has_key('action') and \
077 77               form['action'].value != 'reedit':
078 78           doResults(who, howmany)
079 79      else: <$nopage>
080 80        showForm(who, howmany)
081 81      else: <$nopage>
082 82        showError(error)
083 83
084 84 if __name__ == '__main__':
085 85      process()
086  <$nopage>
```

`friends3.py` is not too unlike `friends2.py`. We invite the reader to compare the differences; we present a brief summary of the major changes for you here:

### Abridged line-by-line explanation

### Line 8

We take the URL out of the form because we now need it in two places, the results page being the new customer.

### Lines 10 – 19, 69 – 71, 75 – 82

All of these lines deal with the new feature of having an error screen. If the user does not select a radio button indicating the number of friends, the `howmany` field is not passed to the server. In such a case, the `showError()` function returns the error page to the user.

The error page also features a JavaScript "Back" button. Because buttons are input types, we need a form, but no action is needed because we are simply just going back one page in the browsing history. Although our script currently supports (a.k.a. detects, tests for) only one type of error, we still use a generic `error` variable in case we wanted to continue development of this script to add more error detection in the future.

### Lines 27, 38–41, 49, and 52–55

One goal for this script is to create a meaningful link back to the form page from the results page. This is implemented as a link to give the user the ability to return to a form page to update the data her or she entered, in case it was erroneous. The new form page makes sense only if it contains information pertaining to the data that has already been entered by the user. (It is frustrating for users to reenter their information from scratch!)

To accomplish this, we need to embed the current values into the updated form. In line 27, we add a value for the name. This value will be inserted into the name field, if given. Obviously, it will be blank on the initial form page. In Line 38–41, we set the radio box which corresponds to the number of friends currently chosen. Finally, on lines 49 and the updated `doResults()` function on lines 52–55, we create the link with all the existing information which "returns" the user to our modified form page.

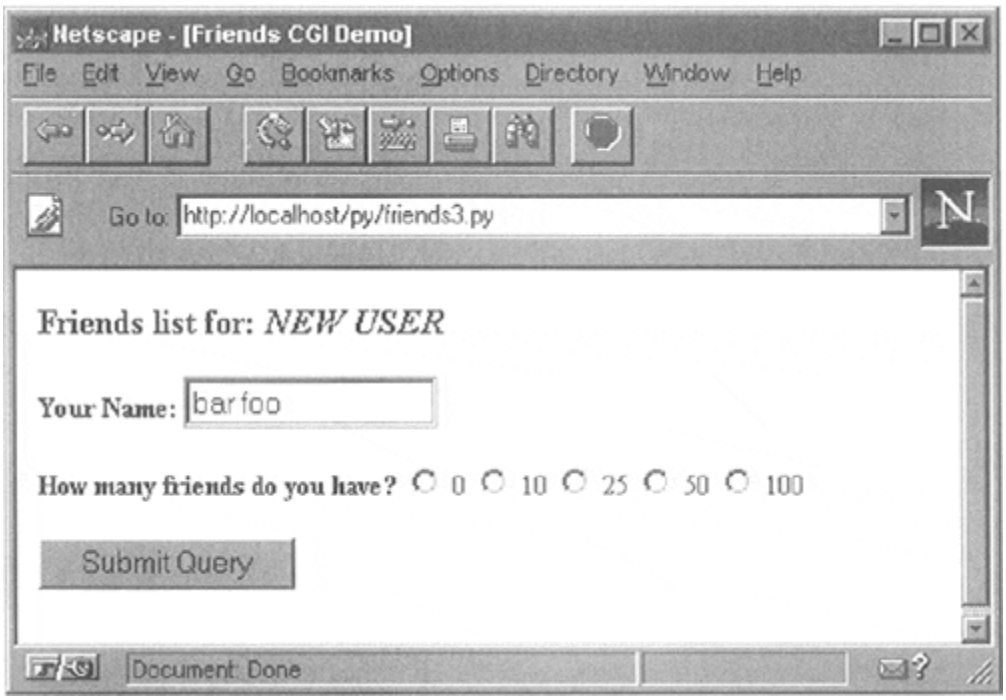
### Line 62

Finally, we added a simple feature that we thought would add a nice aesthetic touch. In the screens for `friends1.py` and `friends2.py`, the text entered by the user as his or her name is taken verbatim. You will notice in the screens above that if the user does not capitalize his or her names, that is reflected in the results page. We added a call to the `string.capwords()` function to automatically capitalize a user's name. The `capwords()` function will capitalize the first letter of each word in the string that is passed in. This may or may not be a desired feature, but we thought that we would share it with you so that you know that such functionality exists.

We will now present four screens which shows the progression of user interaction with this CGI form and script.

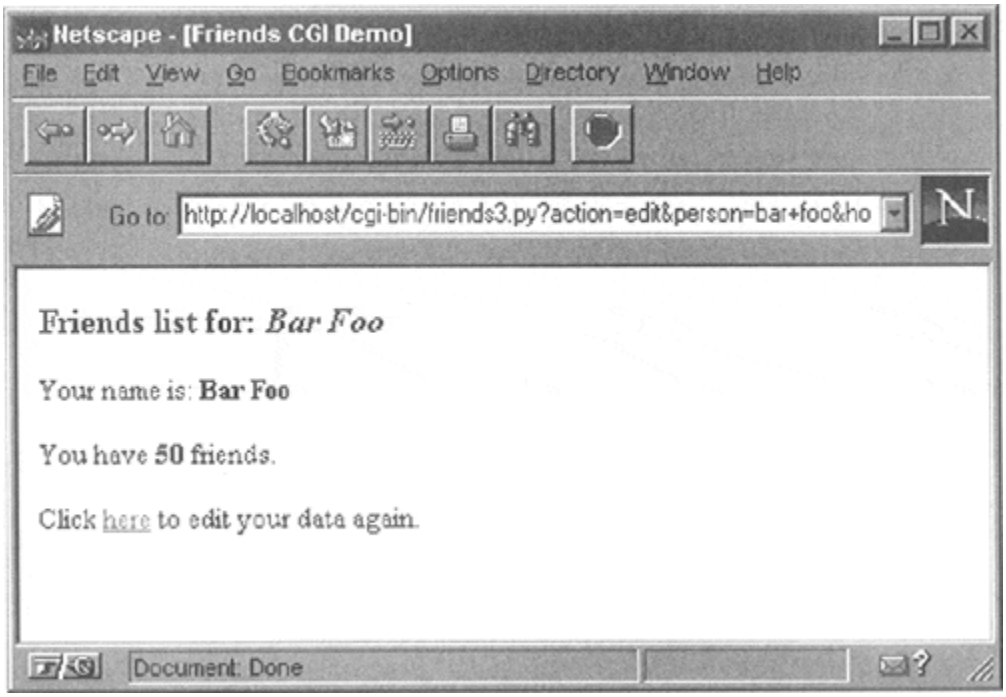
In the first screen, shown in Figure19-9, we invoke `friends3.py` to bring up the now-familiar form page. We enter a name "bar foo," but deliberately avoid checking any of the radio buttons. The resulting error after submitting the form can be seen in the second screen (Figure19-10).

**Figure 19-9. Friends Initial Form Page in Netscape3 on Windows**




We click on the "Back" button, check the "50" radio button, and resubmit our form. The results page, seen in Figure 19-11, is also familiar, but now has an extra link at the bottom. This link will take us back to the form page.

**Figure 19-11. Friends Results Page (Valid Input)**



The only difference between the new form page and our original is that all the data filled in by the user is now set as the "default" settings, meaning that the values are already available in the form. We can see this in Figure19-12.

**Figure 19-10. Friends Error Page (invalid user input)**

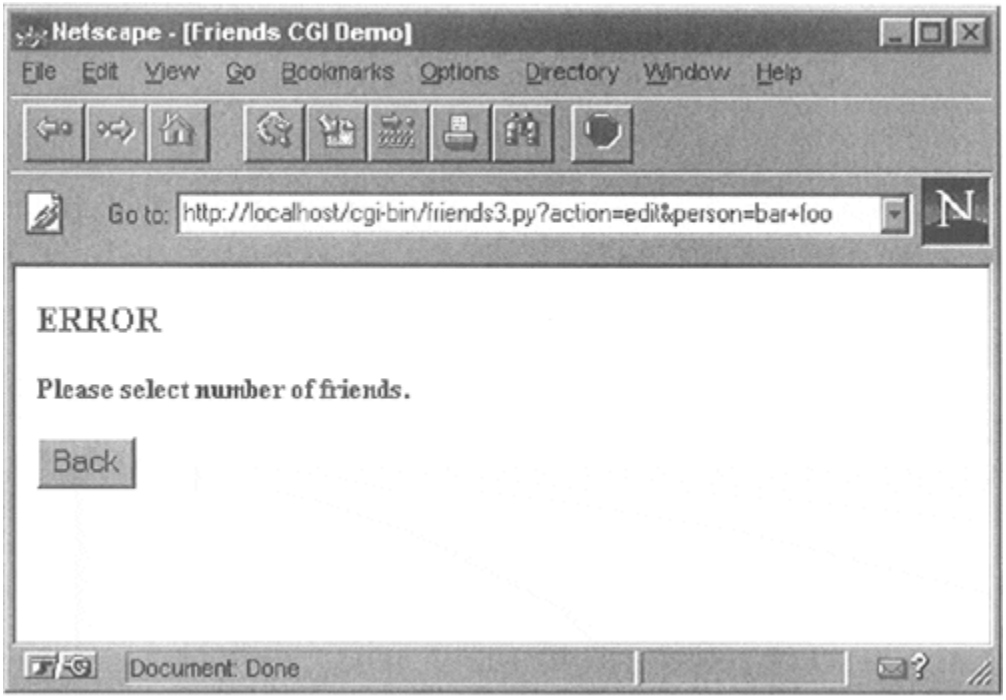
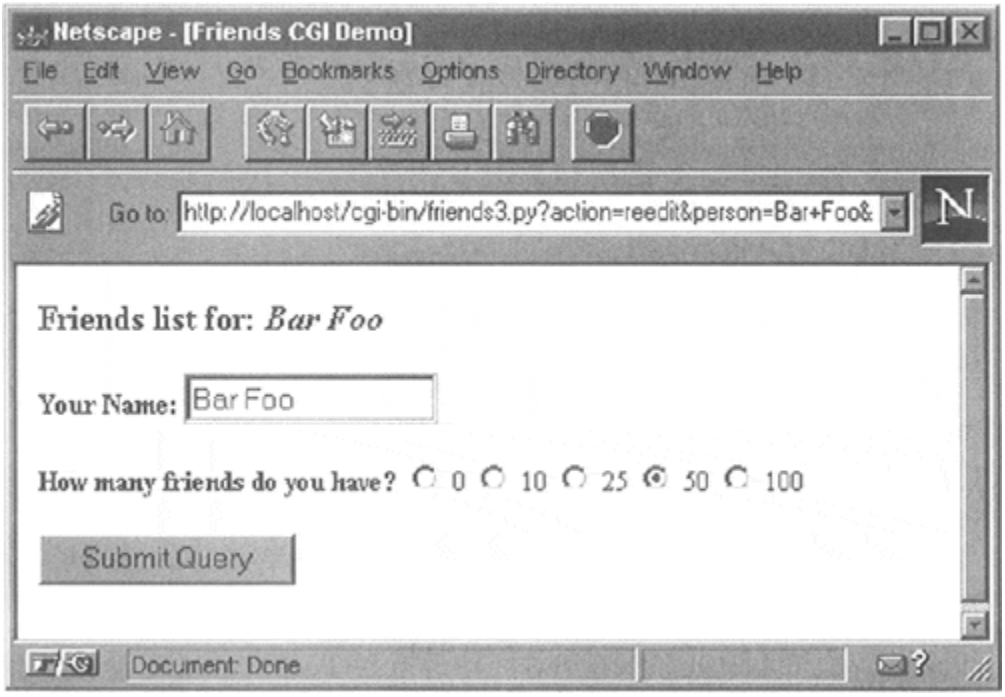



**Figure 19-12. Friends Updated Form Page with Current Information**



Now the user is able to make changes to either of the fields and resubmit their form.

You will no doubt begin to notice that as our forms and data get more complicated, so does the generated HTML, especially for complex results pages. If you ever get to a point where generating the HTML text is interfering with your application, you may consider connecting with a Python module such as `HTMLgen`, an external Python module which specializes in HTML generation.

## Advanced CGI

We will now take a look at some of the more advanced aspects of CGI programming. These include: the use of *cookies*—cached data saved on the client side, multiple values for the same CGI field and file upload using multipart form submissions. To save space, we will show you all three of these features with a single application. Let's take a look at multipart submissions first.

### Multipart Form Submission and File Uploading

Currently, the CGI specifications only allow two types of form encodings, "application/x-www-form-urlencoded" and "multipart/form-data." Because "application/x-www-form-urlencoded" is the default, there is never a need to state the encoding in the FORM tag like this:

```
<FORM enctype="application/x-www-form-urlencoded" …>
```

But for multipart forms, you must explicitly give the encoding as:

```
<FORM enctype="multipart/form-data" …>
```

You can use either type of encoding for form submissions, but at this time, file uploads can only be performed with the multipart encoding. Multipart encoding was invented by Netscape in the early days but since has been adopted by Microsoft (starting with version 4 of Internet Explorer) as well as other browsers.

File uploads are accomplished using the file input type:

```
<INPUT type=file name=…>
```

This directive presents an empty text field with a button on the side which allows you to browse your file directory structure for a file to upload. On most browsers, this button

says "Browse," but your mileage may vary. (For example, we will be using the Opera browser in our examples which has a button labeled with ellipses "…".)

When using multipart, your Web client's form submission to the server will look amazingly like (multipart) e-mail messages with attachments. A separate encoding was needed because it just wouldn't be necessarily wise to "urlencode" a file, especially a binary file. The information still gets to the server, but is just "packaged" in a different way.

Regardless of whether you use the default encoding or the multipart, the `cgi` module will process them in the same manner, providing keys and corresponding values in the form submission. You will simply access the data through your `FieldStorage` instance as before.

## Multivalued Fields

In addition for file uploads, we are also going to show you how to process fields with multiple values. The most common case is when you have a set of checkboxes allowing a user to select from various choices. Each of the checkboxes is labeled with the same field name, but to differentiate them, each will have a different value associated with a particular checkbox.

As you know, the data from the user is sent to the server in key-value pairs during form submission. When more than one checkbox is submitted, you will have multiple values associated with the same key. In these cases, rather than being given a single `MiniFieldStorage` instance for your data, the `cgi` module will create a list of such instances which you will iterate over to obtain the different values. Not too painful at all.

## Cookies

Finally, we will use cookies in our example. If you are not familiar with cookies, they are just bits of data information which a server at a Web site will request to be saved on the client side, e.g., the browser.

Because HTTP is a "stateless" protocol, information that has to be carried from one page to another can be accomplished by using key-value pairs in the request as you have seen in the GET requests and screens earlier in this chapter. Another way of doing it, as we have also seen before, is using hidden form fields, such as the action variable in some of the later `friends*.py` scripts. These variables and their values are managed by the server because the pages they return to the client must embed these in generated pages.

One alternative to maintaining persistency in state across multiple page views is to save the data on the client side instead. This is where cookies come in. Rather than embedding data to be saved in the returned Web pages, a server will make a request to the client to save a cookie. The cookie is linked to the domain of the originating server (so a server cannot set nor override cookies from other Web sites) and has an expiration date (so your browser doesn't become cluttered with cookies).

These two characteristics are tied to a cookie along with the key-value pair representing the data item of interest. There are other attributes of cookies such as a domain subpath or a request that a cookie should only be delivered in a secure environment.

By using cookies, we no longer have to pass the data from page to page to track a user. Although they have been subject to a good amount of controversy over the privacy issue, most Web sites use cookies responsibly. To prepare you for the code, a Web server requests a client store a cookie by sending the "Set-Cookie" header immediately before the requested file.

Once cookies are set on the client side, requests to the server will automatically have those cookies sent to the server using the `HTTP_COOKIE` environment variable. The cookies are delimited by semicolons and come in "key=value" pairs. All your application needs to do to access the data values is to split the string several times (i.e., using `string.split()` or manual parsing). The cookies are delimited by semicolons ( ; ), and each key-value pair is separated by equal signs ( = ).

Like multipart encoding, cookies originated from Netscape, who implemented cookies and wrote up the first specification which is still valid today. You can access this document at the following Web site:

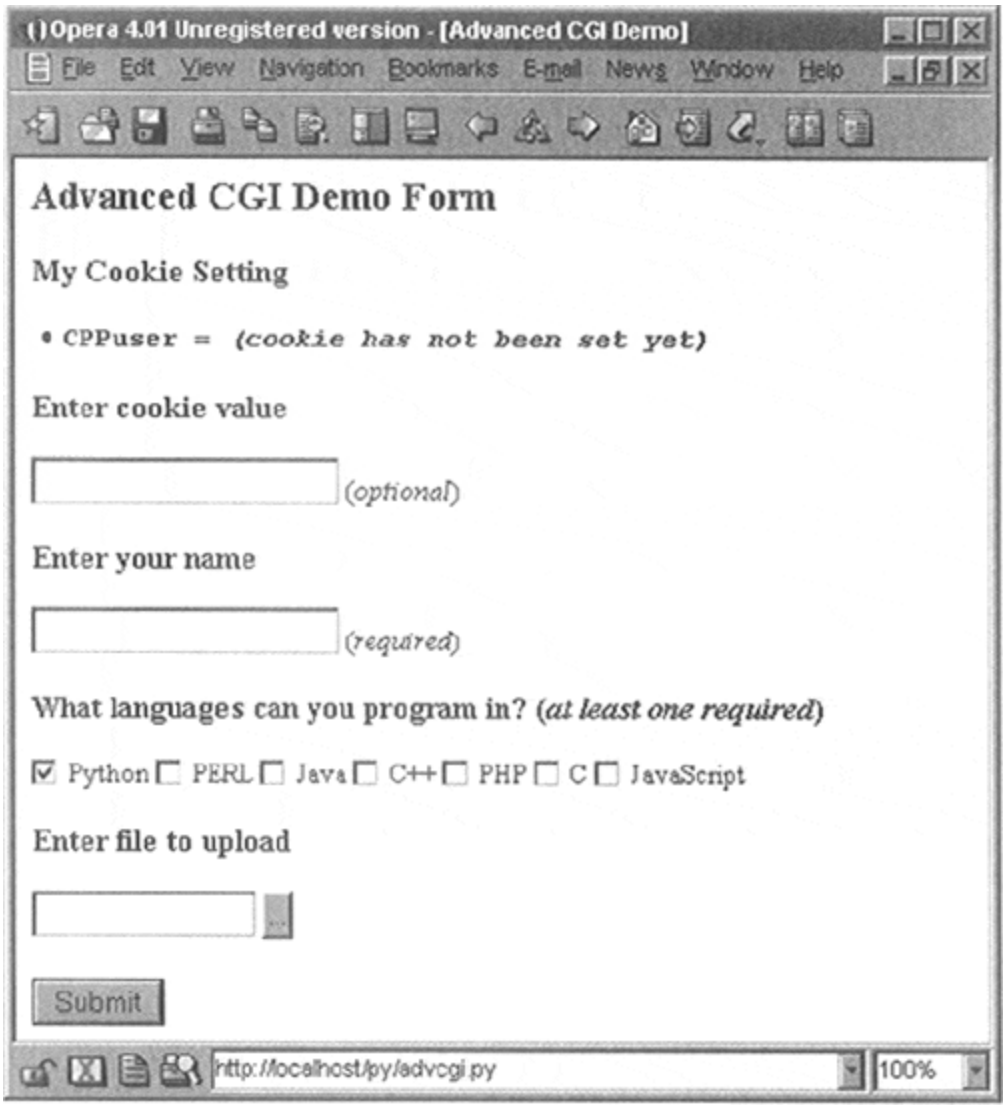
http://www.netscape.com/newsref/std/cookie_spec.html

Once cookies are standardized and this document finally obsoleted, you will be able to get more current information from Request for Comment documents (RFCs). The most current one for cookies at the time of publication is RFC 2109.

## Using Advanced CGI

We now present our CGI application, `advcgi.py`, which has code and functionality not too unlike the `friends3.py` script seen earlier in this chapter. The default first page is a user fill-out form consisting of four main parts: user-set cookie string, name field, checkbox list of programming languages, and file submission box. An image of this screen can be seen in Figure19-13, this time using the Opera 4 browser in a Windows environment.

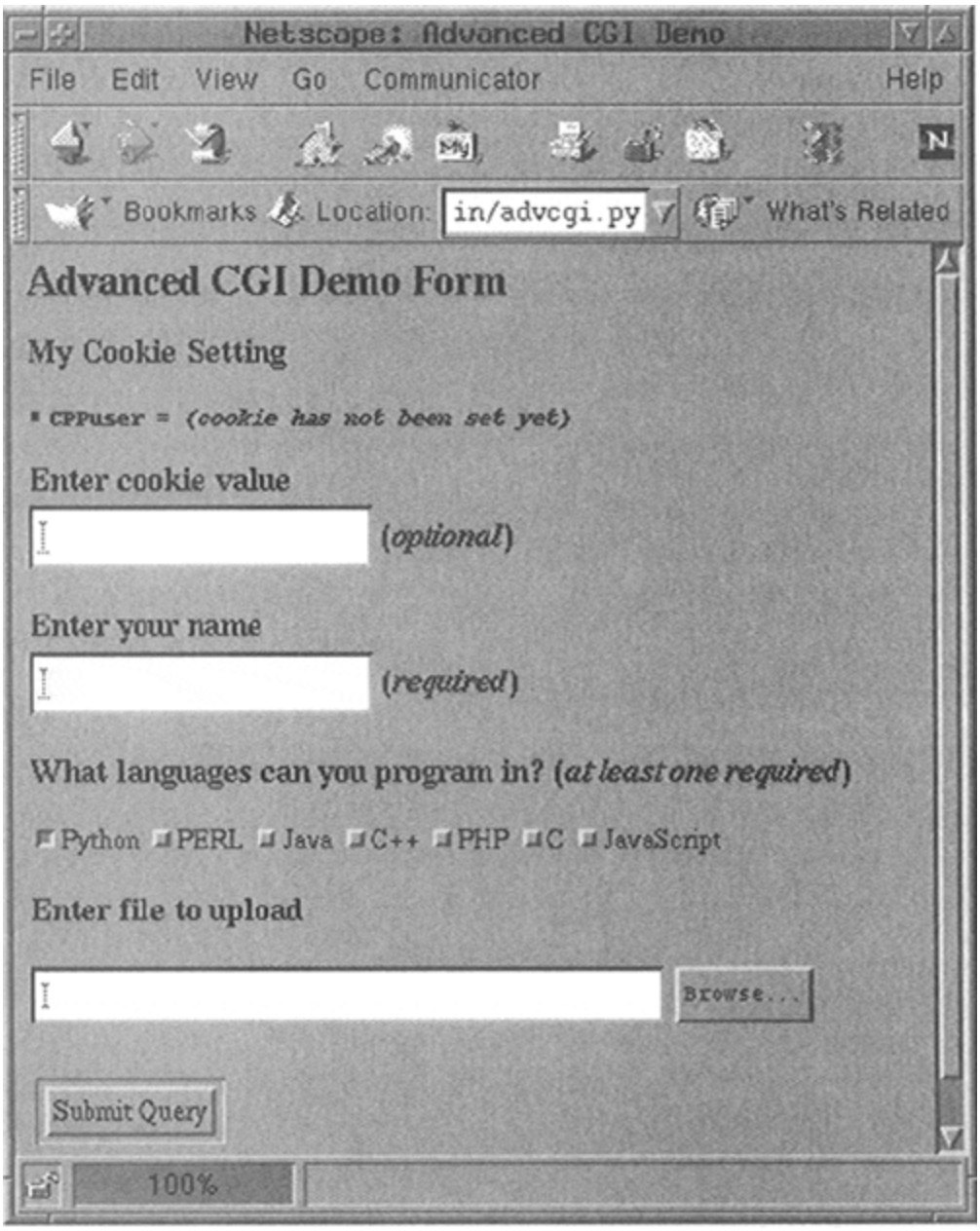
**Figure 19-13. Upload and Multivalue Form Page in Opera4 on Windows**




In a browser world dominated by the Netscape and Microsoft browsers, we seldom hear of others such as Opera and Lynx, but they are out there! Opera, in particular, is known to have excellent footprint (memory size) and speed characteristics.

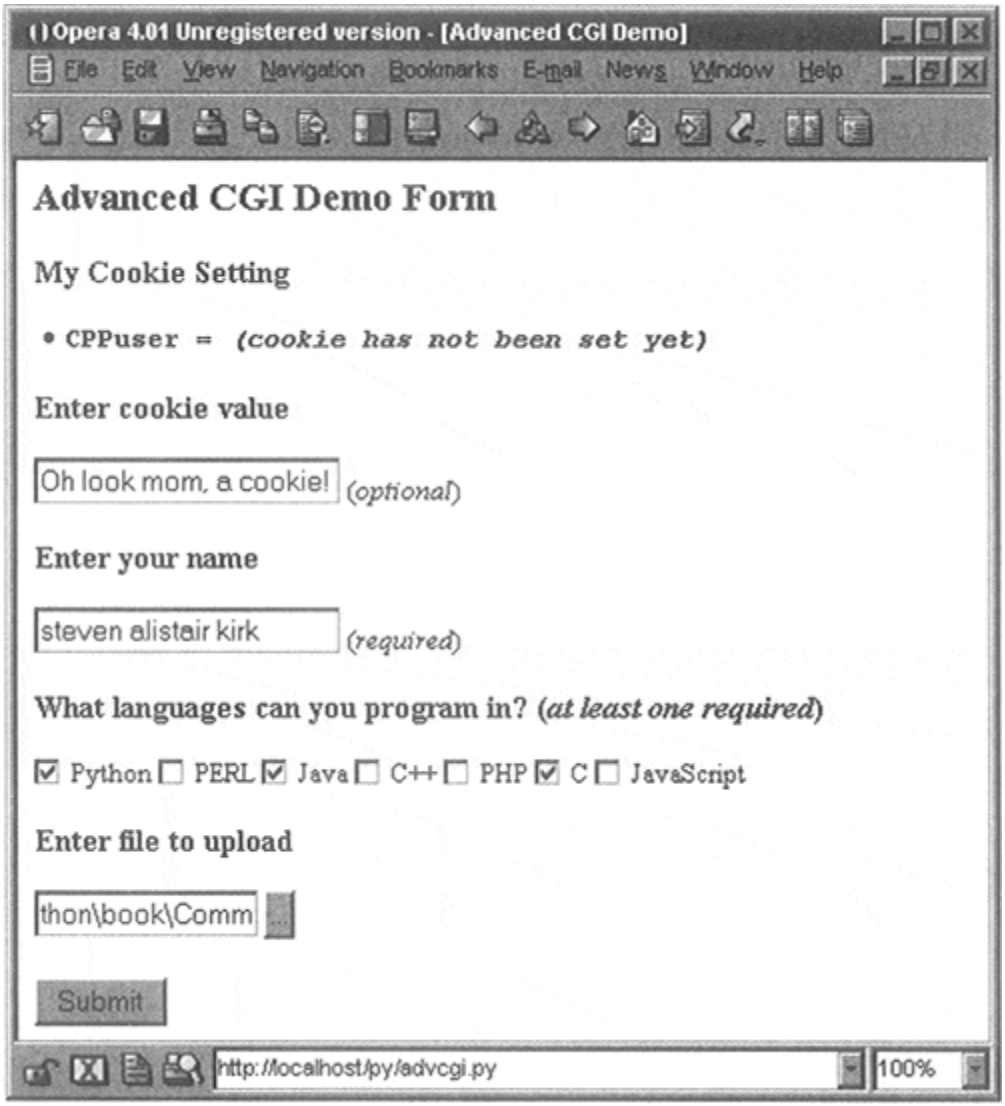
Well, just so you aren't totally uncomfortable, let's take a peek at what the same form looks like from Netscape running on Linux, as in Figure19-14. As you can see, Netscape uses "Browse" as the file upload label instead of the ellipses. (The rest of the screens for this section will feature Opera.)

**Figure 19-14. The Same Advanced CGI Form but in Netscape4 on Linux**




From this form, we can enter our information, such as the sample data given in Figure19-15.

**Figure 19-15. One Possible Form Submission in our Advanced CGI Demo**

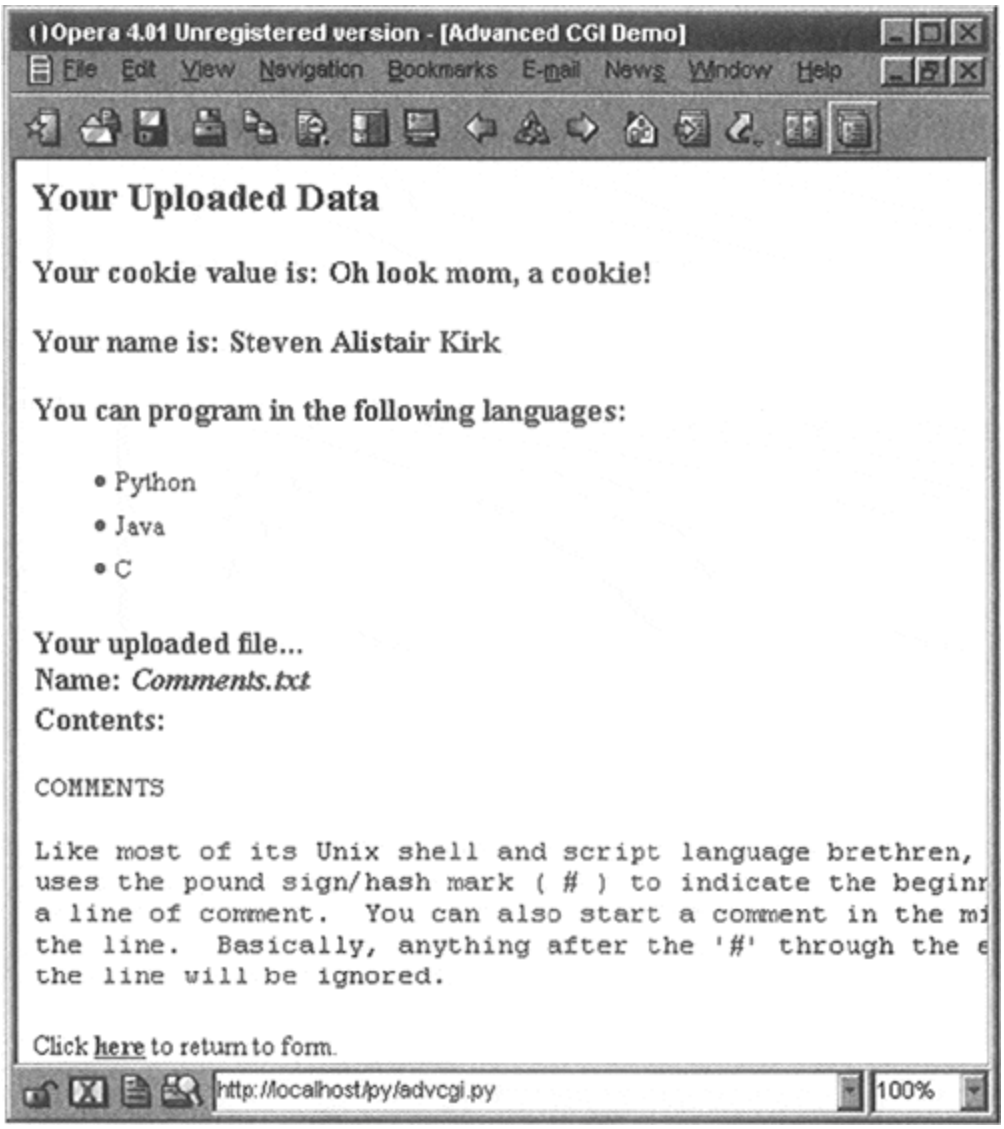



The data is submitted to the server using multipart encoding and is retrieved in the same manner on the server side using the `FieldStorage` instance. The only tricky part is in retrieving the uploaded file. In our application, we choose to iterate over the file, reading it line-by-line. It is also possible to read in the entire contents of the file if you are not wary of its size.

Since this is the first occasion data is received by the server, it is at this time, when returning the results page back to the client, that we use the "Set-Cookie:" header to cache our data in browser cookies.

In Figure19-16, you will see the results after submitting our form data. All the fields the user entered are shown on the page. The contents of the filename given in the final dialog box was actually uploaded to the server and displayed as well.

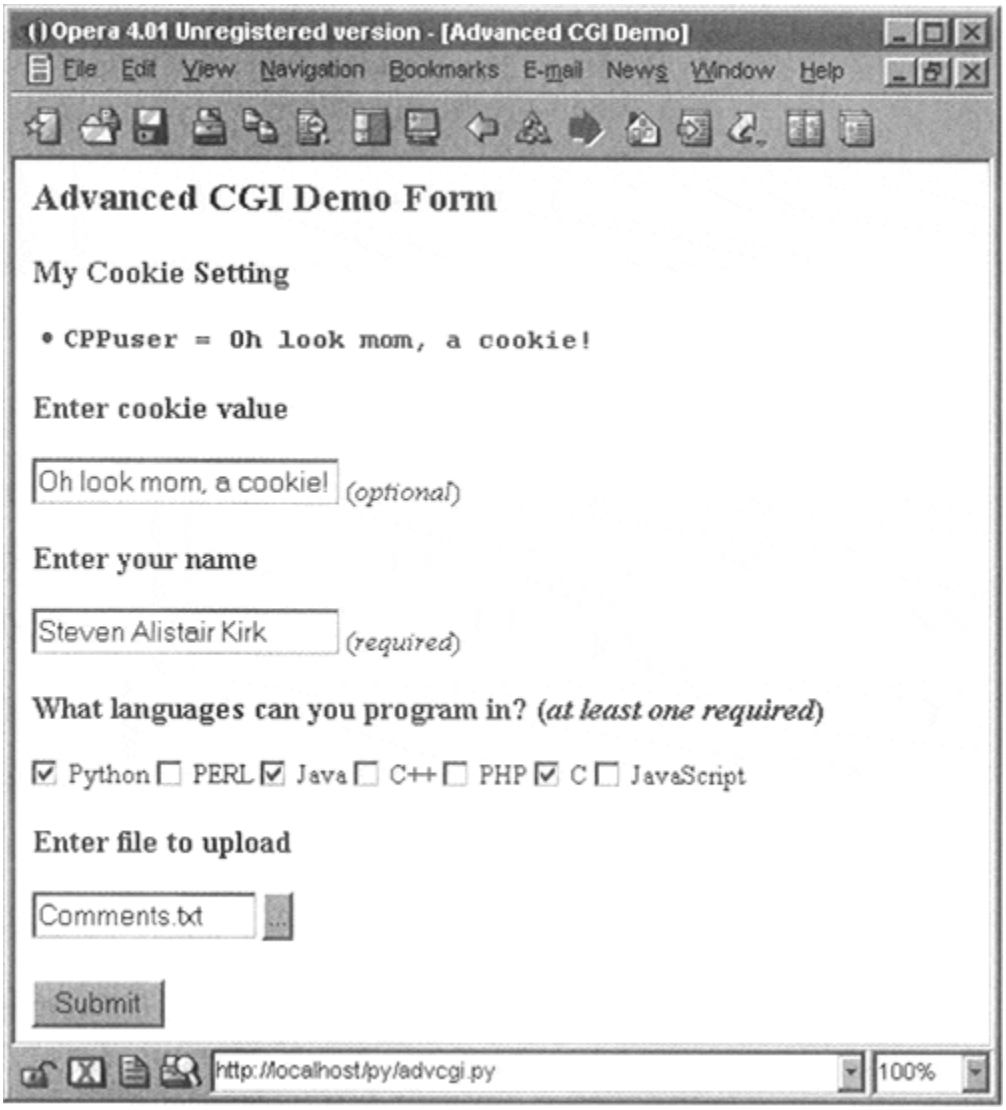
**Figure 19-16. Results Page Generated and Returned by the Web Server**




You will also notice the link at the bottom of the results page which returns us to the form page, again using the same CGI script.

If we click on that link at the bottom, no form data is submitted to our script, causing a form page to be displayed. Yet, as you can see from Figure 19-17, what shows up is anything but an empty form! Information previously entered by the user shows up! How did we accomplish this with no form data (either hidden or as query arguments in the URL)? The secret is that the data is stored on the client side in cookies, two in fact.

**Figure 19-17. Form Page With Data Loaded from the Client Cookies**




The user cookie holds the string of data typed in by the user in the "Enter cookie value" form field, and the user's name, languages they are familiar with, and uploaded file are stored in the info cookie.

When the script detects no form data, it shows the form page, but before the form page has been created, it grabs the cookies from the client (which are automatically transmitted by the client when the user clicks on the link) and fills out the form accordingly. So when the form is finally displayed, all the previously entered information appears to the user like magic.

We are sure you are eager to take a look at this application, so here it is, presented in Example 19-6.

**Example 19.6. Advanced CGI Application (`advcgi.py`)**

The crawler has one main class which does everything, `AdvCGI`. It has methods to show either form, error, or results pages as well as those which read or write cookies from/to the client (a web browser).

```
 <$nopage>
001 1    #!/usr/bin/env python
002 2
003 3    from cgi import FieldStorage
004 4    from os import environ
005 5    from cStringIO import StringIO
006 6    from urllib import quote, unquote
007 7    from string import capwords, strip, split, join
008 8
009 9    class AdvCGI:
010 10
011 11       header = 'Content-Type: text/html\n\n'
012 12       url = '/py/advcgi.py'
013 13
014 14       formhtml = '''<HTML><HEAD><TITLE>
015 15   Advanced CGI Demo</TITLE></HEAD>
016 16   <BODY><H2>Advanced CGI Demo Form</H2>
017 17   <FORM METHOD=post ACTION="%s ENCTYPE="multipart/form-data">
018 18   <H3>My Cookie Setting</H3>
019 19   <LI> <CODE><B>CPPuser = %s</B></CODE>
020 20   <H3>Enter cookie value<BR>
021 21   <INPUT NAME=cookie value="%s"> (<I>optional</I>)</H3>
022 22   <H3>Enter your name<BR>
023 23   <INPUT NAME=person VALUE="%s"> (<I>required</I>)</H3>
024 24   <H3>What languages can you program in?
025 25   (<I>at least one required</I>)</H3>
026 26   %s
027 27   <H3>Enter file to upload</H3>
028 28   <INPUT TYPE=file NAME=upfile VALUE="%s" SIZE=45>
029 29   <P><INPUT TYPE=submit>
030 30   </FORM></BODY></HTML>'''
031 31
032 32       langSet = ('Python', 'PERL', 'Java', 'C++', 'PHP',
033 33                  'C', 'JavaScript')
034 34       langItem = \
035 35           '<INPUT TYPE=checkbox NAME=lang VALUE="%s"%s> %s\n'
036 36
037 37       def getCPPCookies(self):# read cookies from client
038 38           if environ.has_key('HTTP_COOKIE'):
039 39               for eachCookie in map(strip, \
040 40                      split(environ['HTTP_COOKIE'], ';')):
041 41                   if len(eachCookie) > 6 and \
042 42                         eachCookie[:3] == 'CPP':
043 43                      tag = eachCookie[3:7]
044 44                      try: <$nopage>
045 45                         self.cookies[tag] = \
046 46                            eval(unquote(eachCookie[8:]))
047 47                      except (NameError, SyntaxError):
048 48                          self.cookies[tag] = \
049 49                              unquote(eachCookie[8:])
050 50           else: <$nopage>
051 51               self.cookies['info'] = self.cookies['user'] = ''
```

```
052 52
053 53          if self.cookies['info'] != '':
054 54              self.who, langStr, self.fn = \
055 55                  split(self.cookies['info'], ':')
056 56              self.langs = split(langStr, ',')
057 57          else: <$nopage>
058 58              self.who = self.fn = ' '
059 59              self.langs = ['Python']
060 60
061 61      def showForm(self):            # show fill-out form
062 62          self.getCPPCookies()
063 63          langStr = ''
064 64          for eachLang in AdvCGI.langSet:
065 65              if eachLang in self.langs:
066 66                  langStr = langStr + AdvCGI.langItem % \
067 67                      (eachLang, ' CHECKED', eachLang)
068 68              else: <$nopage>
069 69                  langStr = langStr + AdvCGI.langItem % \
070 70                      (eachLang, '', eachLang)
071 71
072 72             if not self.cookies.has_key('user') or \
073 73                  self.cookies['user'] == '':
074 74              cookStatus = '<I>(cookie has not been set
yet)</I>'
075 75              userCook = ''
076 76          else: <$nopage>
077 77              userCook = cookStatus = self.cookies['user']
078 78
079 79          print AdvCGI.header + AdvCGI.formhtml % (AdvCGI.url,
080 80              cookStatus, userCook, self.who, langStr, self.fn)
081 81
082 82      errhtml = '''<HTML><HEAD><TITLE>
083 83  Advanced CGI Demo</TITLE></HEAD>
084 84  <BODY><H3>ERROR</H3>
085 85  <B>%s</B><P>
086 86  <FORM><INPUT TYPE=button VALUE=Back
087 87  ONCLICK="window.history.back()"></FORM>
088 88  </BODY></HTML>'''
089 89
090 90      def showError(self):
091 91           print AdvCGI.header + AdvCGI.errhtml % (self.error)
092 92
093 93      reshtml =    '''<HTML><HEAD><TITLE>
094 94  Advanced CGI Demo</TITLE></HEAD>
095 95  <BODY><H2>Your Uploaded Data</H2>
096 96  <H3>Your cookie value is: <B>%s</B></H3>
097 97  <H3>Your name is: <B>%s</B></H3>
098 98  <H3>You can program in the following languages:</H3>
099 99  <UL>%s</UL>
100 100 <H3>Your uploaded file…<BR>
101 101 Name: <I>%s</I><BR>
102 102 Contents:</H3>
103 103 <PRE>%s</PRE>
104 104 Click <A HREF="%s"><B>here</B></A> to return to form.
105 105 </BODY></HTML>'''
106 106
107 107     def setCPPCookies(self):# tell client to store cookies
```

```
108 108          or eachCookie in self.cookies.keys():
109 109              print 'Set-Cookie: CPP%s=%s; path=/' % \
110 110                  (eachCookie, quote(self.cookies[eachCookie]))
111 111
112 112      def doResults(self):# display results page
113 113          MAXBYTES = 1024
114 114          langlist = ''
115 115          for eachLang in self.langs:
116 116            langlist = langlist + '<LI>%s<BR>' % eachLang
117 117
118 118          filedata = ''
119 119          while len(filedata) < MAXBYTES:# read file chunks
120 120              data = self.fp.readline()
121 121              if data == '': break <$nopage>
122 122              filedata = filedata + data
123 123          else:                  # truncate if too long
124 124              filedata = filedata + \
125 125              '… <B><I>(file truncated due to size)</I></B>'
<$nopage>
126 126          self.fp.close()
127 127          if filedata == '':
128 128          filedata = \
129 129      '<B><I>(file upload error or file not given)</I></B>'
<$nopage>
130 130      filename = self.fn
131 131
132 132      if not self.cookies.has_key('user') or \
133 133               self.cookies['user'] == '':
134 134         cookStatus = '<I>(cookie has not been set yet)</I>'
<$nopage>
135 135          userCook = ''
136 136      else: <$nopage>
137 137          userCook = cookStatus = self.cookies['user']
138 138
139 139      self.cookies['info'] = join([self.who, \
140 140          join(self.langs, ','), filename], ':')
141 141      self.setCPPCookies()
142 142      print AdvCGI.header + AdvCGI.reshtml % \
143 143               (cookStatus, self.who, langlist,
144 144               filename, filedata, AdvCGI.url)
145 145
146 146  def go(self):# determine which page to return
147 147      self.cookies = {}
148 148      self.error = ''
149 149      form = FieldStorage()
150 150      if form.keys() == []:
151 151           self.showForm()
152 152           return <$nopage>
153 153
154 154      if form.has_key('person'):
155 155           self.who = capwords(strip(form['person'].value))
156 156           if self.who == '':
157 157                self.error = 'Your name is required. (blank)'
158 158      else: <$nopage>
159 159           self.error = 'Your name is required. (missing)'
160 160
161 161      if form.has_key('cookie'):
```

```
162 162          self.cookies['user'] = unquote(strip(\
163 163                           form['cookie'].value))
164 164      else: <$nopage>
165 165          self.cookies['user'] = ''
166 166
167 167      self.langs = []
168 168      if form.has_key('lang'):
169 169          langdata = form['lang']
170 170          if type(langdata) == type([]):
171 171               for eachLang in langdata:
172 172                    self.langs.append(eachLang.value)
173 173          else: <$nopage>
174 174               self.langs.append(langdata.value)
175 175      else: <$nopage>
176 176          self.error = 'At least one language required.'
177 177
178 178      if form.has_key('upfile'):
179 179          upfile = form["upfile"]
180 180          self.fn = upfile.filename or ''
181 181          if upfile.file:
182 182               self.fp = upfile.file
183 183          else: <$nopage>
184 184               self.fp = StringIO('(no data)')
185 185      else: <$nopage>
186 186               self.fp = StringIO('(no file)')
187 187               self.fn = ''
188 188
189 189      if not self.error:
190 190               self.doResults()
191 191      else: <$nopage>
192 192               self.showError()
193 193
194 194  if __name__ == '__main__':
195 195      page = AdvCGI()
196 196      page.go()
197  <$nopage>
```

`advcgi.py` looks strikingly similar to our `friends3.py` CGI scripts seen earlier in this chapter. It has form, results, and error pages to return. In addition to all of the advanced CGI features which are part of our new script, we are also using more of an object-oriented feel to our script by using a class with methods instead of just a set of functions. The HTML text for our pages are now static data for our class, meaning that they will remain constant across all instances—even though there is actually only one instance in our case.

### Line-by-line (Block-by-block) explanation

### Lines 1 – 7

The usual start-up and import lines appear here. The only module you may not be familiar with is `cStringIO`, which we briefly introduced at the end of Chapter 10 and also used in Example 19-1. `cStringIO.StringIO()` creates a file-like object out of a

string so that access to the string is similar to opening a file and using the handle to access the data.

**Lines 9 – 12**

After the `AdvCGI` class is declared, the `header` and `url` (static class) variables are created for use by the methods displaying all the different pages.

**Lines 14 – 80**

All the code in this block is used to generate and display the form page. The data attributes speak for themselves. `getCPPCookies()` obtains cookie information sent by the Web client, and `showForm()` collates all the information and sends the form page back to the client.

**Lines 82 – 91**

This block of code is responsible for the error page.

**Lines 93 – 144**

The results page is created using this block of code. The `setCPPCookies()` method requests that a client store the cookies for our application, and the `doResults()` method puts together all the data and sends the output back to the client.

**Lines 146 – 196**

The script begins by instantiating an `AdvCGI` page object, then call its `go()` method to start the ball rolling, in contrast to a strictly procedural programming process. The `go()` method contains the logic that reads all incoming data and decides which page to show.

The error page will be displayed if no name was given or if no languages were checked. The `showForm()` method is called to output the form if no input data was received, and the `doResults()` method is invoked otherwise to display the results page.

Handling the person field is the same as we have seen in the past, a single key-value pair; however, collecting the language information is a bit trickier since we must check for either a `(Mini)FieldStorage` instance or a list of such instances. We will employ the familiar `type()` built-in function for this purpose. In the end, we will have a list of a single language name or many, depending on the user's selections.

The use of cookies to contain data illustrates how they can be used to avoid using any kind of CGI field pass-through. You will notice in the code which obtains such data that no CGI processing is invoked, meaning that the data does not come from the `FieldStorage` object. The data is passed to us by the Web client with each request and the values (user's chosen data as well as information to fill in a succeeding form with pre-existing information) are obtained from cookies.

Because the `showResults()` method receives the new input from the user, it has the responsibility of setting the cookies, i.e., by calling `setCPPCookies()`. `showForm()` however, must read in the cookies' values in order to display a form page with the current user selections. This is done by its invocation of the `getCPPCookies()` method.

Finally, we get to the file upload processing. Regardless of whether a file was actually uploaded, `FieldStorage` is given a file handle in the file attribute. If the value attribute is accessed, then entire contents of the file will be placed into value. As a better alternative, you can access the file pointer—the file attribute—and perhaps read only one line at a time or other kind of slower processing.

In our case, file uploads are only part of user submissions, so we simply pass on the file pointer to the `doResults()` function to extract the data from the file. `doResults()` will display only the first 1K of the file for space reasons and to show you that it is not necessary (or necessarily productive/useful) to display a four megabyte binary file.

## Web (HTTP) Servers

Until now, we have been discussing the use of Python in creating Web clients and performing tasks to aid Web servers in CGI request processing. We know (and have seen earlier in Sections 19.2 and 19.3) that Python can be used to create both simple and complex Web clients. Complexity of CGI requests goes without saying.

However, we have yet to explore the creation of Web *servers,* and that is the focus of this section. If the Netscape, IE, Opera, Mozilla, and Lynx browsers are among the most popular Web clients, then what are the most common Web servers? They are Apache, Netscape, and IIS. In situations where these servers may be overkill for your desired application, we would like to use Python to help us create simple yet useful Web servers.

### Creating Web Servers in Python

Since you have decided on building such an application, you will naturally be creating all the custom stuff, but all the base code you will need is already available in the Python Standard Library. To create a Web server, a base server and a "handler" are required.

The base (Web) server is a boilerplate item, a must have. Its role is to perform the necessary HTTP communication between client and server. The base server is (appropriately) named `HTTPServer` and is found in the `BaseHTTPServer` module.

The handler is the piece of software which does the majority of the "Web serving." It processes the client request and returns the appropriate file, whether static or dynamically-generated by CGI. The complexity of the handler determines the complexity of your Web server. The Python standard library provides three different handlers.

The most basic, plain, vanilla handler, named `BaseHTTPRequestHandler`, is found in the `BaseHTTPServer` module, along with the base Web server. Other than taking a client

request, no other handling is implemented at all, so you have to do it all yourself, such as in our `myhttpd.py` server below.

The `SimpleHTTPRequestHandler,` available in the `SimpleHTTP-Server` module, builds on `BaseHTTPRequestHandler` by implementing the standard GET and HEAD requests in a fairly straightforward manner. Still nothing sexy, but it gets the simple jobs done.

Finally, we have the `CGIHTTPRequestHandler,` available in the `CGIHTTPServer` module, which takes the `SimpleHTTPRequestHandler` and adds support for POST requests. It has the ability to call CGI scripts to perform the requested processing and can send the generated HTML back to the client.

The three modules and their classes are summarized in Table 19-6.

**Table 19.6. Web Server Modules and Classes**

| Module | Description |
|---|---|
| `BaseHTTPServer` | provides the base Web server and base handler classes, `HTTPServer` and `BaseHTTPRequestHandler,` respectively |
| `SimpleHTTPServer` | contains the `SimpleHTTPRequestHandler` class to perform GET and HEAD requests |
| `CGIHTTPServer` | contains the `CGIHTTPRequestHandler` class to process POST requests and perform CGI execution |

To be able to understand how the more advanced handlers found in the `SimpleHTTPServer` and `CGIHTTPServer` modules work, we will implement simple GET processing for a `BaseHTTPRequestHandler.` In Example 19-7, we present the code for a fully working Web server, `myhttpd.py.`

This server subclasses `BaseHTTPRequestHandler` and consists of a single `do_GET()` method, which is called when the base server receives a GET request. We attempt to open the path passed in by the client and if present, return an "OK" status (200) and forward the downloaded Web page. If the file was not found, returning a 404 status.

The `main()` function simply instantiates our Web server class and invokes it to run our familiar infinite server loop; shutting it down if interrupted by ^C or similar keystroke. If you have appropriate access and can run this server, you will notice that it displays loggable output which will look something like:

**Example 19.7. Simple Web Server (`myhttpd.py`)**

This simple Web server can read GET requests, fetch a Web page (.html file) and return it to the calling client. It uses the `BaseHTTPRequestHandler` found in `BaseHTTPServer` and implements the `do_GET()` method to enable processing of GET requests

```
 <$nopage>
001 1   #!/usr/bin/env python
002 2
003 3   from os import curdir, sep
004 4   from BaseHTTPServer import \
005 5                BaseHTTPRequestHandler, HTTPServer
006 6
007 7   class MyHandler(BaseHTTPRequestHandler):
008 8
009 9       def do_GET(self):
010 10          try: <$nopage>
011 11              f = open(curdir + sep + self.path)
012 12              self.send_response(200)
013 13              self.send_header('Content-type',
014 14                                       'text/html')
015 15              self.end_headers()
016 16              self.wfile.write(f.read())
017 17              f.close()
018 18          except IOError:
019 19              self.send_error(404, \
020 20                  'File Not Found: %s' % self.path)
021 21
022 22  def main():
023 23      try: <$nopage>
024 24          server = HTTPServer(('', 80), MyHandler)
025 25          print 'Welcome to the machine…',
026 26          print 'Press ^C once or twice to quit.'
027 27          server.serve_forever()
028 28      except KeyboardInterrupt:
029 29          print '^C received, shutting down server'
030 30          server.socket.close()
031 31
032 32  if __name__ == '__main__':
033 33      main()
034  <$nopage>

# myhttpd.py
Welcome to the machine... Press ^C once or twice to quit
localhost - - [26/Aug/2000 03:01:35] "GET /index.html
HTTP/1.0" 200 -
localhost - - [26/Aug/2000 03:01:29] code 404, message
File Not Found: /dummy.html
localhost - - [26/Aug/2000 03:01:29] "GET /dummy.html
HTTP/1.0" 404 -
localhost - - [26/Aug/2000 03:02:03] "GET /hotlist.htm
HTTP/1.0" 200 -
```

Of course, our simple little Web server is so simple, that it cannot even process plain text files. We leave that as an exercise for the reader, which can be found at the end of the chapter.

As you can see, it doesn't take much to have a Web server up and running in pure Python. There is plenty more you can do to enhance the handlers to customize it to your specific application. Please review the Library Reference for more information on these modules (and their classes) discussed in this section.

## Related Modules

In Table 19-7, we present a list of modules which you may find useful for Web and Internet development.

The parsing modules deal with recognizing documents in specific formats.

You can write POP- or IMAP-compliant mail clients using the corresponding protocol modules.

Python has plenty of modules to support most kinds of binary file encoding for e-mail and other MIME-oriented applications.

You can create clients for common Internet protocols like HTTP, FTP, Telnet, and NNTP with the appropriate modules. Be aware that `urllib` provides a high-level interface to protocols supported by your browser such as HTTP and FTP, so use of the lower-level protocol modules only makes sense when you cannot get all you want from `urllib`.

Finally, we have the `HTMLgen` external module and the commercial Zope (Z Object Publishing Environment) system by Digital Creations. We introduced the `HTMLgen` module briefly at the end of Section 19.5. It definitely comes in handy when you need to generate more complex HTML documents via CGI scripts.

**Table 19.7. Web Programming Related Modules**

| Module | Description |
|---|---|
| Parsing | |
| `htmllib` | parses simple HTML files |
| `sgmllib` | parses simple SGML files |

| | |
|---|---|
| xmllib | parses simple XML files |
| robotparser[a] | parses robots.txt files for URL "fetchability" analysis |
| Mail Client Protocols | |
| poplib | use to create POP3 clients |
| imaplib | use to create IMAP4 clients |
| Mail and MIME Processing and Data Encoding Formats | |
| mailcap | parses mailcap files to obtain MIME application delegations |
| mimetools | provides functions for manipulating MIM-encoded messages |
| mimetypes | provides MIME type associations |
| MimeWriter | generates MIME-encoded multipart files |
| multifile | can parse multipart MIME-encoded files |
| quopri | en-/decodes data using quoted-printable encoding |
| rfc822 | parses RFC822-compliant e-mail headers |
| smtplib | uses to create SMTP (Simple Mail Transfer Protocol) clients |

| | |
|---|---|
| base64 | en-/decodes data using base64 encoding |
| binascii | en-/decodes data using base64, binhex, or uu (modules) |
| binhex | en-/decodes data using binhex4 encoding |
| uu | en-/decodes data using uuencode encoding |
| Internet Protocols | |
| httplib[a] | use to create HTTP (HyperText Transfer Protocol) clients (modified in Python 1.6 to support HTTP 1.1 and SSL) |
| ftplib | use to create FTP (File Transfer Protocol) clients |
| gopherlib | use to create Gopher clients |
| telnetlib | use to create Telnet clients |
| nntplib | use to create NNTP (Network News Transfer Protocol [Usenet]) clients |
| External/Commercial | |
| HTMLgen | use with CGI to generate complex HTML documents |
| Zope (not a module) | web object publishing product and Python Web application development environment (http://www.zope.org) |

[a] new or modified in Python 1.6

Zope is an open source Web publishing and application development platform which has Python code everywhere. Part of it is written in Python, and Python can be used to create extensions to Zope. Although it is in our Related Modules section, Zope is not a specific module as it is a powerful system for Web publishing.

Zope presents an extremely powerful alternative when simple CGI and database access just do not cut it for the application you are trying to build. Material on Zope itself can take up a book's length—you may even see one soon! We invite the reader to explore this system if desiring to create any complex system.

The `robotparser` module is new as of Python 1.6 and the `httplib` and `urllib` modules have been modified for 1.6 to support HTTP connections over SSL. (See Section 19.2.2 for a really brief introduction.) Also, a new module `webbrowser,` was introduced in 2.0 to provide a platform-independent way to launch a Web browser.

## Exercises

> *`urllib` Module and Files.* Update the `friends3.py` script so that it stores names and corresponding number of friends into a 2-column text file on disk and continues to add names each time the script is run.

EXTRA CREDIT: Add code to dump the contents of such a file to the Web browser (in HTML format). Additional EXTRA CREDIT: Create a link that clears all the names in this file.

*`urllib` Module.* Write a program that takes a user-input URL (either a Web page or an FTP file, i.e., http://www.python.org or ftp://ftp.python.org/pub/python/README, and downloads it to your machine with the same filename (or modified name similar to the original if it is invalid on your system). Web pages (HTTP) should be saved as `.htm` or `.html` files, and FTP'd files should retain their extension.

*`urllib` Module.* Rewrite the `grabWeb.py` script of Example 11.2 which downloads a Web page and displays the first and last non-blank lines of the resulting HTML file so that you use `urlopen()` instead of `urlretrieve()` to process the data directly (as opposed to downloading the entire file first before processing it).

*URLs and Regular Expressions.* Your browser may save your favorite Web site URLs as a "bookmarks" HTML file (Netscape browsers do this) or as a set of .URL files in a "favorites" directory (Microsoft browsers do this). Find your browser's method of recording your "hot links" and the location of where and how they stored. Without altering any of the files, strip the URLs and names of the corresponding Web sites (if given) and produce a 2-column list of names and links as output, and storing this data into a disk file. Truncate site names or URLs to keep each line of output within 80 columns in size.

*URLs, `urllib` Module, Exceptions, and REs.* As a follow-up problem to the previous, add code to your script to test each of your favorite links. Report back a list of dead links

(and their names), i.e., Web sites that are no longer active or a Web page that has been removed. Only output and save to disk the still-valid links.

> *Error Checking.* The `friends3.py` script reports an error if no radio button was selected to indicate the number of friends. Update the CGI script to also report an error if no name (e.g., blank or whitespace) is entered.

EXTRA CREDIT: We have so far explored only server-side error checking. Explore JavaScript programming and implement client-side error checking by creating JavaScript code to check for both error situations so that these errors are stopped before they reach the server.

Problems 19–7 to 19–10 below pertain to *Web server access log files and Regular Expressions.* Web servers (and their administrators) generally have to maintain an access log file (usually `logs/access_log` from the main Web server directory) which tracks requests file. Over a period of time, such files get large and either need to be stored or truncated. Why not save only the pertinent information and delete the files to conserve disk space? The exercises below are designed to give you some exercise with REs and how they can be used to help archive and analyze Web server data.

> **Step 19-7.**

Count how many of each type of request (GET vs. POST) exist in the log file.

Count the successful page/data downloads: Display all links which resulted in a return code of 200 (OK [no error]) and how many times each link was accessed.

Count the errors: Show all links which resulted in errors (return codes in the 400s or 500s) and how many times each link was accessed.

Track IP addresses: For each IP address, output a list of each page/data downloaded and how many times that link was accessed.

*Simple CGI.* Create a "Comments" or "Feedback" page for a Web site. Take user feedback via a form, process the data in your script, and return a "thank you" screen.

*Simple CGI.* Create a Web guestbook. Accept a name, an e-mail address, and a journal entry from a user and log it to a file (format of your choice). Like the previous problem, return a "thanks for filling out a guestbook entry" page. Also provide a link which allows users to view guestbooks.

> *Web Browser Cookies and Web Site Registration.* Update your solution to Exercise 13-4. so that your user-password information pertains to Web site registration instead of a simple text-based menu system.

EXTRA CREDIT: familiarize yourself with setting Web browser cookies and maintain a login session for 4 hours from the last successful login.

*Stock Quote Information.* There are many online services which allow users to look up stock quote price information. A few of these sites, such as Yahoo! for example, allow users to download such data in a comma-delimited spreadsheet format. Become familiar with one of these sites and learn how to download stock price information onto your local hard drive. Create a Python application not only to perform the download, but also to be able to read, parse, and display the saved data for a specified set of stock ticker symbols.

EXTRA CREDIT: Integrate your solution to the previous problem by registering users and allowing individual portfolios using the classes created for your solution to Exercise 13-13.

*Stock Quote Information.* Update your solution to the previous problem by bypassing the downloading of the information to a local file. Open a connection directly to a Web server and parse the stock data as it streams down to your application, and display this information to the screen.

**NOTE**

Python on the Windows 32-bit platform contains connectivity to Component Object Model (COM), a Microsoft interfacing technology that allows objects to talk to one another, or more higher-level, applications to talk to one another, without any language- or format-dependence. You can read all about COM in Hammond and Robinson. The combination of Python and COM presents a unique opportunity to create Python scripts which can talk to such applications as Word or Excel.

*Stock Quotes and Excel/COM programming* (  ). Familiarize yourself with COM programming in Python, then use your solution to the previous problem to create a new application which downloads stock quote information and transfers that data directly to an Excel spreadsheet. You may choose to have the user manually invoke the Python script to update the data, or if you have a direct connection to the Internet, have your script update the data periodically during the business day. Merge any element of your solution to the previous problem by providing automatically-updating Excel spreadsheets for multiple portfolios.

*Multithreaded COM Programming* (  ). Update your solution to the previous problem so that the downloads of data happen "concurrently" using multiple threads.

*Web Database Application.* Think of a database schema you want to provide as part of a Web database application. For this multi-user application, you want to provide everyone read access to the entire contents of the database, but perhaps only write access to each individual. One example may be an "address book" for your family and relatives. Each family member, once successfully logged in, is presented with a Web page with several options, add an entry, view my entry, update my entry, remove or delete my entry, and view all entries (entire database).

Design a `UserEntry` class and create a database entry for each instance of this class. You may use any solution created for any previous problem to implement the registration framework. Finally, you make use any type of storage mechanism for your database, either a relational database such as mySQL or some of the simpler Python persistent storage modules such as `anydbm` or `shelve.`

*Electronic Commerce Engine.* Use the classes created for your solution to Exercise 13-11 and add some product inventory to create a potential electronic commerce Website. Be sure your Web application also supports multiple customers and provides registration for each user.

> *Dictionaries and cgi module.* As you know, the `cgi.FieldStorage()` method returns a dictionary-like object containing the key-value pairs of the submitted CGI variables. You can use methods such as `keys()` and `has_key()` for such objects. In Python 1.5, a `get()` method was added to dictionaries which returned the value of the requested key, or the default value for a non-existent key. `FieldStorage` objects do not have such a method. Let's say we grab the form in the usual manner of:
>
> ```
> form = cgi.FieldStorage()
> ```
>
> Add a similar `get()` method to class definition in `cgi.py` (you can rename it to `mycgi.py` or something like that) such that code which looks like this:
>
> ```
> if form.has_key(\qwho\q):
>     who = form[\qwho\q].value
> else:
>     who = \q(no name submitted)\q
> ```
>
> … can be replaced by a single line which makes forms even more like a dictionary:
>
> ```
> howmany = form.get('who', '(no name submitted)')
> ```
>
> *Creating Web Servers.* Our code for `myhttpd.py` in Section 19.7 is only able to read HTML files and return them to the calling client. Add support for plain text files with the ".txt" ending. Be sure that you return the correct MIME type of "text/plain."

EXTRA CREDIT: add support for JPEG files ending with either ".jpg" or ".jpeg" and having a MIME type of "image/jpeg".

*Advanced Web Clients.* Update the `crawl.py` script in Section 19.3 to also download links which use the "`ftp:`" scheme. All "`mailto:`" links are ignored by `crawl.py.` Add support to ensurethat it also ignores "`telnet:, news:, gopher:,`" and "`about:`" links.

*Advanced Web Clients.* The `crawl.py` script in Section 19.3 only downloads `.html` files via links found in Web pages at the same site and does not handle/save images which are also valid "files" for those pages. It also does not handle servers which are susceptible to URLs which are missing the trailing slash ( `/` ). Add a pair of classes to `crawl.py` to deal with these problems. A `My404UrlOpener` class should subclass `urllib.FancyURLOpener` and consist of a single method, **`http_error_404()`** which determines if a 404 error was reached using a URL without a trailing slash. If so, it adds the slash and retries the request again (and only once). If it still fails, return a real 404 error. You must set `urllib._urlopener` with an instance of this class so that `urllib` uses it.

Create another class called `LinkImageParser` which derives from `htmllib.HTMLParser`. This class should contain a constructor to call the base class constructor as well as initialize a list for the image files parsed from Web pages. The `handle_image()` method should be overridden to add image filenames to the image list (instead of discarding them like the current base class method does).

# Chapter 20. Extending Python

Chapter Topics

Introduction/Motivation

Extending Python

Create Application Code

Wrap Code in Boilerplate

Compile

Import and Test

Related Topics

In this chapter, we will discuss how we can take code written externally and integrate that functionality into your Python programming environment. We will first give you motivation for doing such a thing, then take you through the step-by-step process on how to do it. We should point out, though, that because extensions are primarily done in the C language, all of the example code you will see in this section is pure C.

## Introduction/Motivation

### What Are Extensions?

In general, any code that you write that can be integrated or imported into another Python script can be considered an "extension." This new code can be written in pure Python or in a compiled language like C and C++ (or Java for JPython). However, a more "strict" definition of an extension is relegated to the latter category, the topic of this chapter.

One great feature of Python is that its extensions interact with the interpreter in exactly the same way as the regular Python modules. Python was designed so that the abstraction of module import hides the underlying implementation details from the code which uses such extensions. Unless the client programmer searches the file system, he or she simply cannot tell whether a module is written in Python or in a compiled language.

**NOTE**

We will note here that extensions are generally available in a development environment where you compile your own Python interpreter. There is a subtle relationship between manual compilation versus obtaining the binaries. Although compilation may be a bit

trickier than just downloading and installing binaries, you have the most flexibility in customizing the version of Python you are using.

If you intend to create extensions, you should perform this task in a similar environment. The examples in this chapter use a Unix system (which, by default, comes with compilers), but, assuming you do have access to aC/C++ (or Java) compiler and a Python development environment in C/C++ (or Java), the only differences are in your compilation method. The actual code to make your extensions usable in the Python world is the same on any platform.

## Why Extend Python?

Throughout the brief history of software engineering, programming languages have always been taken at face value. What you see is what you get; it was impossible to add new functionality to an existing language. In today's programming environment however, the ability to customize one's programming environment is now a desired feature; it also promotes code reuse. Languages such as TCL and Python are among the first languages to provide the ability to extend the base language. So why would you want to extend a language like Python which is already feature-rich? There are several good reasons:

Added/extra (non-Python) functionality

One reason for extending Python is the need to have new functionality not provided by the core part of the language. This can be accomplished in either pure Python or as a compiled extension, but there are certain things such as creating new data types or embedding Python in an existing application.

Bottleneck performance improvement

It is well-known that interpreted languages do not perform as fast as compiled languages due to the fact that translation must happen on-the-fly and during runtime. In general, moving a body of code into an extension will improve overall performance. The problem is that it is sometimes not advantageous if the cost is high in terms of resources.

Percentage-wise, it is a wiser bet to do some simple profiling of the code to identify what the bottlenecks are, and move *those* pieces of code out to an extension. The gain can be seen more quickly and without expending as much in terms of resources.

Keep proprietary source code private

Another important reason to create extensions is due to one side effect of having a scripting language. For all the ease-of-use such languages bring to the table, there really is no privacy as far as source code is concerned because the executable *is* the source code.

Code that is moved out of Python and into a compiled language helps keep proprietary code private because you ship a binary object. Because these objects are compiled, they are not as readily able to be reverse-engineered; thus, the source remains more private. This is key when it involves special algorithms, encryption or software security, etc.

Another alternative to keeping code private is to only ship pre-compiled `.pyc` files only. It serves as a good middle ground between releasing the actual source (`.py` files) and having to migrate that code to extensions.

## Extending Python by Writing Extensions

Creating extensions for Python involve three main steps:

Create application code

Wrap code with boilerplates

Compilation

In this section, we will break out all three pieces and expose it all to you.

## Create Your Application Code

First, before any code becomes an extension, create a standalone "library." In other words, create your code keeping in mind that it is going to turn into a Python module. Design your functions and objects with the vision that Python code will be communicating and sharing data with your C code and vice versa.

Next, create test code to bulletproof your software. You may even use the "Pythonic" development method of designating your `main()` function in C as the testing application so that if your code is compiled, linked, and loaded into an executable (as opposed to just a shared object), that invocation of such an executable will result in a regression test of your software library. For our extension example below, this is exactly what we do.

The test case involves two C functions which we want to bring to the world of Python programming. The first is the recursive factorial function, `fac()`. The second, `reverse()`, is a simple string reverse algorithm, whose main purpose is to reverse a string "in place," that is, to return a string whose characters are all reversed from their original positions, all without allocating a separate string to copy in reverse order. Because this involves the use of pointers, we need to carefully design and debug our code before bringing Python into the picture.

Our first version, `Extest1.c`, is presented in Example 20.1.

**Example 20.1. Pure C Version of Library(`Extest1.c`)**

The following code represents our library of C functions which we want to wrap so that we can use this code from within the Python interpreter. `main()` is our tester function.

```
 <$nopage>
001 1  #include <stdio.h>
002 2  #include <stdlib.h>
003 3  #include <string.h>
004 4
005 5  int fac(int n)
006 6  {
007 7      if (n < 2) return(1);
008 8      return((n)*fac(n-1));
009 9  }
010 10
011 11 char *reverse(char *s)
012 12 {
013 13    register char t,
014 14             *p = s,
015 15             *q = (s + (strlen(s) - 1));
016 16
017 17    while (s && (p < q))
018 18    {
019 19        t = *p;
020 20        *p++ = *q;
021 21        *q-- = t;
022 22    }
023 23    return s;
024 24 }
025 25
026 26 void main()
027 27 {
028 28    char s[BUFSIZ];
029 29    printf("4! == %d\n", fac(4));
030 30    printf("8! == %d\n", fac(8));
031 31    printf("12! == %d\n", fac(12));
032 32    strcpy(s, "abcdef");
033 33    printf("reversing 'abcdef', we get '%s'\n", \
034 34        reverse(s));
035 35    strcpy(s, "madam");
036 36    printf("reversing 'madam', we get '%s'\n", \
037 37        reverse(s));
038 38 }
039  <$nopage>
```

This code consists of a pair of functions, `fac()` and `reverse()`, which are implementations of the functionality we described above. `fac()` takes a single integer argument and recursively calculates the result, which is eventually returned to the caller once it exits the outermost call.

The last piece of code is the required `main()` function. We use it to be our tester, sending various arguments to `fac()` and `reverse()`. With this function, we can actual tell whether our code works (or not).

Now we should compile the code. For many versions of Unix with the `gcc` compiler, we use the following command:

```
% gcc Extest1.c -o Extest
%
```

To run our program, we issue the following command and get the output:

```
% Extest
4! == 24
8! == 40320
12! == 479001600
reversing 'abcdef', we get 'fedcba'
reversing 'madam', we get 'madam'
%
```

We stress again that you should try to complete your code as much as possible, because you do not want to mix debugging of your library with potential bugs when integrating with Python. In other words, keep the debugging of your core code separate from the debugging of the integration. The closer you write your code to Python interfaces, the sooner your code will be integrated and work correctly.

Each of our functions takes a single value and returns a single value. It's pretty cut and dried, so there shouldn't be a problem integrating with Python. Note that so far, there no connection or relationship with Python as of now. We are simply creating a standard C or C++ application.

## Wrap Your Code in Boilerplate

The entire implementation of an extension primarily revolves around the "wrapping" concept which we introduced earlier in Section 13.15.1. You should design your code in such a way that there is a smooth transition between the world of Python and your implementing language. This interfacing code is commonly called "boilerplate" code because it is a necessity if your code is to talk to the Python interpreter.

There are 4 main pieces to the boilerplate software:

Include Python header file

Add `PyObject* Module_func()` Python wrappers for each module function

Add `PyMethodDef ModuleMethods[]` array/table for each module function

Add **`void`** `initModule()` module initializer function

### Include Python header file

The first thing you should do is to find out where your Python include files are and make sure your compiler has access to that directory, which is usually `/usr/local/include/python1.x,` or `/usr/include/python1.x,` where the "1.x" is your version of Python. (It is probably 1.5 or 2.0.) If you compiled and installed your Python interpreter, then you shouldn't have a problem because the system generally knows where your files are installed.

Add the inclusion of the `Python.h` header file to your source. The line will look something like:

```
#include "Python.h"
```

That's the easy part. Now you have to add the rest of the boilerplate software.

### Add `PyObject* Module_func()` Python wrappers for each function

This part is the trickiest. For each function which you want accessible to the Python environment, you will create a **`static`** `PyObject*` function with the module name (along with an underscore [ _ ]) prepended to it.

For example, we want `fac()` to be one of the functions available for import from Python and will use Extest as the name of our final module, so we create a "wrapper" called `Extest_fac().` So in the client Python script, there will be an "**`import`** `Extest`" and an "`Extest.fac()`" call somewhere (or just "`fac()`" for "**`from`** `Extest` **`import`** `fac`").

The job of the wrapper is to take Python values, convert them to C, then make a call to the appropriate function with what we want. When our function has completed, and it is time to return to the world of Python, it is also the job of this wrapper to take whatever return values we designate, convert them to Python, and then perform the return, passing back any values as necessary.

In the case of `fac(),` when the client program invokes `Extest.fac(),` our wrapper will be called. We will accept a Python integer, convert it to a C integer, call our C function `fac()` and obtain another integer result. We then have to take that return value, convert it back to a Python integer, then return from the call. (In your head, try to keep in mind that you are writing the code that will proxy for a "**`def`** `fac(n)`" declaration. When you are returning, it is as if that imaginary Python `fac()` function is completing.)

So, you're asking, how does this conversion take place? The answer is with the `PyArg_Parse*()` functions when going from C to Python, and `Py_BuildValue()` when returning from C to Python.

The `PyArg_Parse*()` functions are similar to the C `sscanf()` function. It takes a stream of bytes, and, according to some format string, parcels them off to corresponding container variables, which, as expected, take pointer addresses. They both return 1 on successful parsing and 0 otherwise.

`Py_BuildValue()` works like `sprintf()`, taking a format string and converting all arguments to a single returned object containing those values in the formats that you requested.

You will find a summary of these functions below in Table 20.1:

**Table 20.1. Converting Data Between Python and C/C++**

| *Function* | *Description* |
|---|---|
| ***C to Python*** | |
| **int** `PyArg_ParseTuple()` | converts (a tuple of) arguments passed from Python to C |
| **int** `PyArg_ParseTupleAndKeywords()` | same as `PyArg_ParseTuple()` but also parses keyword arguments |
| ***Python to C*** | |
| `PyObject*` `Py_BuildValue()` | converts C data values into a Python return object, either a single object or a single tuple of objects |

A set of conversion codes is used to convert data objects between C and Python; they are given in Table 20.2.

**Table 20.2. Common Codes to Convert Data Between Python and C/C++**

| *Format Code* | *Python Type* | *C/C++ Type* |
|---|---|---|
| s | string | **char*** |
| z | string/None | **char*c**/NULL |
| i | int | **int** |
| l | long | **long** |
| c | string | **char** |
| d | float | **double** |
| D | complex | Py_Complex* |
| O | (any) | PyObject* |
| S | string | PyStringObject |

These conversion codes are the ones given in the respective format strings that dictate how the values should be converted when moving between both languages. NOTE: the

conversion types are different for Java since all data types are classes. Consult the JPython documentation to obtain the corresponding Java types for Python objects.

Here we show you our completed `Extest_fac()` wrapper function:

```
static PyObject *
Extest_fac(PyObject *self, PyObject *args) {

     int res;              // parse result
     int num;              // arg for fac()
     PyObject* retval;     // return value

     res = PyArg_ParseTuple(args, "i", &num);
     if (!res)             // TypeError
         return NULL;
     }
     res = fac(num);
     retval = (PyObject*)Py_BuildValue("i", res);
     return retval;
}
```

The first step is to parse the data received from Python. It should be a regular integer, so we use the "i" conversion code to indicate as such. If the value was indeed an integer, then it gets stored in the num variable. Otherwise, `PyArg_ParseTuple()` will return a NULL, in which case we also return one. In our case, it will generate a `TypeError` exception that tells the client user that we are expecting an integer.

We then call `fac()` with the value stored in `num` and put the result in `res,` reusing that variable. Now we build our return object, a Python integer, again using a conversion code of "i". `Py_BuildValue()` creates an integer Python object which we then return. That's all there is to it!

In fact, once you have created wrapper after wrapper, you tend to shorten your code somewhat to avoid extraneous use of variables. Try to keep your code legible, though. We take our `Extest_fac()` function and reduce it to its smaller version given here, using only one variable, `num:`

```
static PyObject *
Extest_fac(PyObject *self, PyObject *args) {
    int num;
    if (!PyArg_ParseTuple(args, "i", &num)) return NULL;
        return (PyObject*)Py_BuildValue("i", fac(num));
}
```

What about `reverse()`? Well, since you already know how to return a single value, we are going to change our `reverse()` example somewhat, returning two values instead of

one. We will return a pair of strings as a tuple, the first element being the string as passed in to us, and the second being the newly-reversed string.

To show you that there is some flexibility, we will call this function `Extest.doppel()` to indicate that its behavior differs from `reverse()`. Wrapping our code into an `Extest_doppel()` function, we get:

```
static PyObject *
Extest_doppel(PyObject *self, PyObject *args) {
    char *orig_str;
    if (!PyArg_ParseTuple(args, "s", &orig_str)) return NULL;
    return (PyObject*)Py_BuildValue("ss", orig_str, \
        reverse(strdup(orig_str)));
}
```

As in `Extest_fac()`, we take a single input value, this time a string, and store it into `orig_str`. Notice that we use the "s" conversion code now. We then call `strdup()` to create a copy of the string. (Since we want to return the original one as well, we need a string to reverse, so the best candidate is just a copy of the string.) `strdup()` creates and returns a copy which we immediate dispatch to `reverse()`. We get back a reversed string.

As you can see, `Py_BuildValue()` puts together both strings using a conversion string of "ss." This creates a tuple of two strings, the original string and the reversed one. End of story, right? Unfortunately, no.

We got caught by one of the perils of C programming: the memory leak, that is, when memory is allocated but not freed. Memory leaks are analogous to borrowed books from the library but not returning them. You should always release resources which you have acquired when you no longer require them. How did we commit such a crime with our code (which looks innocent enough)?

When `Py_BuildValue()` puts together the Python object to return, it makes copies of the data it has been passed. In our case here, that would be a pair of strings. The problem is that we allocated the memory for the second string, but we did not release that memory when we finished, leaking it. What we really want to do is to build the return object and then free the memory that we allocated in our wrapper. We have no choice but to lengthen our code to:

```
static PyObject *
Extest_doppel(PyObject *self, PyObject *args) {
    char *orig_str;                 // original string
    char *dupe_str;                 // reversed string
    PyObject* retval;

    if (!PyArg_ParseTuple(args, "s", &orig_str)) return NULL;
    retval = (PyObject*)Py_BuildValue("ss", orig_str, \
```

```
        dupestr=reverse(strdup(orig_str)));
    free(dupe_str);
    return retval;
}
```

We introduced the `dupe_str` variable to point to the newly-allocated string, built the return object and referenced it to `retval`. Then we `free()` the memory allocated and finally return back to the caller. Now we are done.

### Add `PyMethodDef ModuleMethods[]` array/table for each module function

Now that both of our wrappers are complete, we want to list them somewhere so that the Python interpreter knows how to import and access them. This is the job of the `ModuleMethods[]` array.

It is made up of an array of arrays, with each individual array containing information about each function, terminated by a NULL array marking the end of the list. For our `Extest` module, we create the following `ExtestMethods[]` array:

```
static PyMethodDef
ExtestMethods[] = {
     { "fac", Extest_fac, METH_VARARGS },
     { "doppel", Extest_doppel, METH_VARARGS },
     { NULL, NULL },
};
```

The Python-accessible names are given, followed by the corresponding wrapping functions. The constant `METH_VARARGS` is given, indicating a set of arguments in the form of a tuple. If we are using `PyArg_ParseTupleAndKeywords()` with keyworded arguments, we would logically OR this flag with the `METH_KEYWORDS` constant. Finally, a pair of NULLs properly terminates our list of two functions.

### Add `void initModule()` module initializer function

The final piece to our puzzle is the module initializer function. This code is called when our module is imported for use by the interpreter. In this code, we make one call to `Py_InitModule()` along with the module name and the name of the `ModuleMethods[]` array so that the interpreter can access our module functions. For our `Extest` module, our `initExtest()` procedure looks like this:

```
    void initExtest() {
          Py_InitModule("Extest", ExtestMethods);
    }
```

We are now done with all our wrapping. We add all this code to our original code from `Extest1.c` and merge the results into a new file called `Extest2.c`, concluding the development phase of our example.

Another approach to creating an extension would be to make your wrapping code first, using "stubs" or test or dummy functions which will, during the course of development, be replaced by the fully functional pieces of implemented code. That way you can ensure that your interface between Python and C is correct, and then use Python to test your C code.

## Compilation

Now we are on to the compilation phase. In order to get your new wrapper Python extension to build, you need to get it to compile with the Python library. This task has finally been standardized across platforms to make life a lot easier for extension designers.

Copy `Misc/Makefile.pre.in`

Create `Setup`

Create `Makefile`

Compile and link your code by running `make`

Import your module from Python

Test function

### Copy `Misc/Makefile.pre.in`

The first step is to copy the `Makefile.pre.in` file from the `Misc` directory of the Python distribution to your local directory where your extension is to be compiled. In fact, all steps take place in this same directory or folder.

### Create `Setup`

The next step is to create a `Setup` file. The first line should contain the string "`*shared*.`" It is followed by line after line of module names followed by source files and compiler options which need to come together to build the module. If you have only one module, then it should be only one line. The format of these lines is the following:

```
modName modFile[1, ***modFile2…][compiler_opts][linker_opts]
```

So for our `Extest` example, our `Setup` file consists of the following pair of lines:

```
*shared*
Extest Extest.2c
```

The "*shared*" string at the top of the Setup file means to create a shared library (.so object file), i.e., Extest.so. This file can then be imported by any Python module just as if your module was written in pure Python.

### Create Makefile

Now we need to create the Makefile. We do this by issuing the make command:

```
% make -f Makefile.pre.in boot
```

This step usually gives a good amount of output, most of which is not important to you. It basically takes the information provided in the Setup file, adds its knowledge of where all the Python files are, and generates a Makefile so that you can build your module object file.

### Compile and link your code by running make

```
% make
gcc -fpic  -O2 -m486 -fno-strength-reduce -I/usr/
include/python1.5 -I/usr/include/python1.5 -
DHAVE_CONFIG_H -c ./Extest2.c
gcc -shared  Extest2.o  -o Extestmodule.so
```

#### NOTE

If your module consists of a single file of the same name, then your shared object file will be the same name, but with a .so extension, i.e., if our module is Extest and our file is Extest.c, then our shared object file would be called Extest.so. If there is more than one file or if there is a single file with a different name, then your module will have a "module" suffix after its name, i.e., Extestmodule.so. In either case, you still import the module by its original name (without the "module").

### Import your module from Python

Now we can test out our module from the interpreter:

```
>>> import Extest
>>> Extest.fac(5)
120
>>> Extest.fac(9)
362880
>>> Extest.doppel('abcdefgh')
('abcdefgh', 'hgfedcba')
>>> Extest.doppel("Madam, I'm Adam.")
("Madam, I'm Adam.", ".madA m'I ,madaM")
```

### Test function

The one last thing we want to do is to add a test function. In fact, we already have one, in the form of the `main()` function. Now it is potentially dangerous to have a `main()` function in our code because there should only be one `main()` in the system. We remove this danger by changing the name of our `main()` to `test()` and wrapping it, adding `Extest_test()` and updating `ExtestMethods` array so that they both look like this:

```
static PyObject *
Extest_test(PyObject *self, PyObject *args) {
    test();
    return (PyObject*)Py_BuildValue("");
}
static PyMethodDef
ExtestMethods[] = {
    { "fac", Extest_fac, METH_VARARGS },
    { "doppel", Extest_doppel, METH_VARARGS },
    { "test", Extest_test, METH_VARARGS },
    { NULL, NULL },
};
```

The `Extest_test()` module function just runs `test()` and returns an empty string, resulting in a Python value of `None` being returned to the caller.

Now we can run the same test from Python:

```
>>> Extest.test()
4! == 24
8! == 40320
12! == 479001600
reversing 'abcdef', we get 'fedcba'
reversing 'madam', we get 'madam'
>>>
```

Below, we present the final version of `Extest2.c` (Example 20.2) that was used to generate the output we just witnessed.

**Example 20.2. Python-wrapped Version of C Library (`Extest2.c`)**

<$nopage>

```
001 1  #include <stdio.h>
002 2  #include <stdlib.h>
003 3  #include <string.h>
004 4
005 5  int fac(int n)
006 6  {
007 7      if (n < 2) return(1);
008 8      return ((n)*fac(n-1));
009 9  }
010 10
011 11 char *reverse(char *s)
012 12 {
013 13     register char t,
014 14                  *p = s,
015 15                  *q = (s + (strlen(s) - 1));
016 16
017 17     while (s && (p < q))
018 18     {
019 19         t = *p;
020 20         *p++ = *q;
021 21         *q-- = t;
022 22     }
023 23     return(s);
024 24 }
025 25
026 26 void test()
027 27 {
028 28     char s[BUFSIZ];
029 29     printf("4! == %d\n", fac(4));
030 30     printf("8! == %d\n", fac(8));
031 31     printf("12! == %d\n", fac(12));
032 32     strcpy(s, "abcdef");
033 33     printf("reversing 'abcdef', we get '%s'\n", \
034 34         reverse(s));
035 35     strcpy(s, "madam");
036 36     printf("reversing 'madam', we get '%s'\n", \
037 37         reverse(s));
038 38 }
039 39
040 40 #include "Python.h"
041 41
042 42 static PyObject *
043 43 Extest_fac(PyObject *self, PyObject *args)
044 44 {
045 45     int num;
046 46     if (!PyArg_ParseTuple(args, "i", &num))
047 47         return NULL;
048 48     return (PyObject*)Py_BuildValue("i", fac(num));
049 49 }
050 50
```

```
051 51 static PyObject *
052 52 Extest_doppel(PyObject *self, PyObject *args)
053 53 {
054 54     char *orig_str;
055 55     char *dupe_str;
056 56     PyObject* retval;
057 57
058 58     if (!PyArg_ParseTuple(args, "s", &orig_str))
059 59         return NULL;
060 60     retval = (PyObject*)Py_BuildValue("ss", orig_str, \
061 61         dupe_str=reverse(strdup(orig_str)));
062 62     free(dupe_str);
063 63     return retval;
064 64 }
065 65
066 66 static PyObject *
067 67 Extest_test(PyObject *self, PyObject *args)
068 68 {
069 69     test();
070 70     return (PyObject*)Py_BuildValue("");
071 71 }
072 72
073 73 static PyMethodDef
074 74 ExtestMethods[] =
075 75 {
076 76     { "fac", Extest_fac, METH_VARARGS },
077 77     { "doppel", Extest_doppel, METH_VARARGS },
078 78     { "test", Extest_test, METH_VARARGS },
079 79     { NULL, NULL },
080 80 };
081 81
082 82 void initExtest()
083 83 {
084 84     Py_InitModule("Extest", ExtestMethods);
085 85 }
086  <$nopage>
```

In this example, we chose to segregate our C code from our Python code. It just kept things easier to read and is no problem with our short example. In practice, these source files tend to get large, and some choose to implement their wrappers completely in a different source file, i.e., `ExtestWrappers.c` or something of that nature.

## Reference Counting

You may recall that Python uses reference counting as a means of keeping track of objects and deallocating objects no longer referenced as part of the garbage collection mechanism. When creating extensions, you must pay extra special attention to how you manipulate Python objects because you must be mindful of whether or not you need to change the reference count for such objects.

There are two types of references you may have to an object, one of which is an owned *reference,* meaning that the reference count to the object is incremented by one to

indicate your ownership. One place where you would definitely have an owned reference is where you create a Python object from scratch.

When you are done with a Python object, you must dispose of your ownership, either by decrementing the reference count, transferring your ownership by passing it on, or storing the object. Failure to dispose of an owned reference creates a memory leak.

You may also have a *borrowed reference* to an object. Somewhat lower on the responsibility ladder, this is where you are passed the reference of an object, but otherwise do not manipulate the data in any way nor do you have to worry about its reference count, so long as you do not hold onto this reference after its reference count has decreased to zero. You may convert your borrowed reference to an owned reference simply by incrementing an object's reference count.

Python provides a pairs of C macros which are used to change the reference count to a Python object. They are given in Table 20.3:

**Table 20.3. Macros for Performing Python Object Reference Counting**

| *Function* | *Description* |
|---|---|
| `Py_INCREF(obj)` | increment the reference count to *obj* |
| `Py_DECREF(obj)` | decrement the reference count to *obj* |

In our above `Extest_test()` function, we return `None` by building a `PyObject` with an empty string; however, it can also be accomplished by becoming an owner of the `None` object, `PyNone`, incrementing your reference count to it, and returning it explicitly, as in the following alternative piece of code:

```
static PyObject *
Extest_test(PyObject *self, PyObject *args) {
    test();
    Py_INCREF(Py_None);
    return PyNone;
}
```

`Py_INCREF()` and `Py_DECREF()` also have versions which check for NULL objects, and they are `Py_XINCREF()` and `Py_XDECREF()`, respectively.

We strongly urge the reader to consult the Python documentation regarding extending and embedding Python for all the details with regards to reference counting (see the documentation reference in the Appendix).

## Threading and GIL Awareness

Extension writers must be aware that their code may be executed in a multithreaded Python environment. Back in Section 17.3.1, we introduced the Python Virtual Machine

(PVM) and the Global Interpreter Lock (GIL) and described how only one thread of execution can be running at any given time in the PVM, and that the GIL is responsible for keeping other threads from running. Furthermore, we indicated that code calling external functions such as in extension code would keep the GIL locked until the call returns.

We also hinted that there was a remedy, a way for the extension programmer to release the GIL, for example before performing a system call. This accomplished by "blocking" your code off to where threads may (and may not) run safely using another pair of C macros, `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS`. A block of code bounded by these macros will permit other threads to run.

As with the reference counting macros, we urge you consult with the documentation regarding extending and embedding Python as well as the Python/C API reference manual.

## Related Topics

### SWIG

There is an external tool available called SWIG, which stands for Simplified Wrapper and Interface Generator. It was written by David Beazley, also the author of Python Essential Reference, and is a software tool that can take annotated C/C++ header files and generate wrapped code, ready to compile for Python, Tcl, and Perl. In fact, it is so simple that once you get comfortable using it, you can practically bypass everything we have discussed in this chapter! You can find out more information about SWIG from its main Web site located at the following Web address (URL):

http://www.swig.org

### Embedding

Embedding is another feature which is available in Python. It is the inverse of an extension. Rather than taking C code and wrapping it into Python, you take a C application and wrap a Python interpreter inside it. This has the effect of giving a potentially large, monolithic, and perhaps rigid, proprietary, and/or mission-critical application the power of having an embedded Python interpreter. Once you have Python, well, it's like a whole new ball game.

To conclude, we would like to mention that there are two pieces of official Python documentation related to the material in this chapter, "Embedding and Extending the Python Interpreter" and "Python/C API Reference Manual." Both will be able to fill in the gaps that we left and are available at the Python home page or directly at this link:

http://www.python.org/doc/ext

## Exercises

*Extending Python.* What are some of the advantages of Python extensions?

*Extending Python.* Can you see any disadvantages or dangers from using extensions?

*Writing Extensions.* Obtain or find a C/C++ compiler and write a small program with it to (re)familiarize yourself with C/C++ programming. Find your Python distribution directory and locate the `Misc/Makefile.pre.in` file. Take the program you just wrote and wrap it in Python. Go through the steps necessary to create a shared object. Access that module from Python and test it.

*Porting from Python to C.* Take several of the exercises you did in earlier chapters and port them to C/C++ as extension modules.

*Wrapping C Code.* Find a piece of C/C++ code which you may have done a long time ago, but want to port to Python. Instead of porting, make it an extension module.

*Writing Extensions.* In Exercise 13-3, you created a `dollarize()` function as part of a class to convert a floating point value to a financial numeric string with embedded dollar signs and commas. Create an extension featuring a wrapped `dollarize()` function and integrate a regression testing function, i.e., `test()`, into the module.

*Extending vs. Embedding.* What is the difference between extending and embedding?

# Answers to Selected Exercises

## Chapter 2

**Q:**

Loops and numbers. Create some loops using both while and for.

**A:**

5. loops and numbers

a)

```
i = 0
while i < 11:
    i = i + 1
```

b)

```
for i in range(11):
     pass
```

**Q:**

Conditionals. Detect whether a number is positive, negative, or zero. Try using fixed values at first, then update your program to accept numeric input from the user

**A:**

6. conditionals

```
n = int(raw_input('enter a number: '))
if n < 0:
    print 'negative'
elif n > 0:
    print 'positive'
else:
    print 'zero'
```

**Q:**

Loops and strings. Take a user input string and display string, one character at a time. As in your above solution, perform this task with a while loop first, then with a for loop

**A:**

7.

```
s = raw_input('enter a string: ')

for eachChar in s:
    print eachChar

for i in range(len(s)):
    print s[i]
```

**Q:**

Loops and operators. Create a fixed list or tuple of 5 numbers and output their sum. Then update your program so that this set of numbers comes from user input. As with the problems above, implement your solution twice, once using while and again with for.

**A:**

8.

```
subtot = 0
for i in range(5):
    subtot = subtot + int(raw_input('enter a number: '))
print subtot
```

## Chapter 3

**Q:**

Identifiers. Which of the following are valid Python identifiers? If not, why not? Of the invalid ones, which are keywords?

**A:**

7. identifiers

| | | |
|---|---|---|
| `40XL` | | number |
| `$saving$` | symbol | |
| **`print`** | | kw |
| `0x40L` | | number |
| `big-daddy` | symbol | |
| `2hot2touch` | number | |
| `thisIsn'tAVar` | symbol | |
| **`if`** | | kw |

| counter-1 | symbol | |
|---|---|---|

## Chapter 4

**Q:**

Object Equality. What do you think is the difference between the expressions type(a) == type(b) and type(a) is type(b)?

**A:**

6. difference between `type(a) == type(b)` and `type(a) is type(b):`

`type(a) == type(b)` whether the value of type(a) is the same as the value of type(b)… == is a value compare

`type(a) is type(b)` whether the type objects returned by type(a) and type(b) are the same object

## Chapter 5

**Q:**

Geometry. Calculate the area and volume of

**A:**

8.

```
import math

def sqcube():
     s= float(raw_input('enter length of one side: '))
     print 'the area is:', s ** 2., '(units squared)'
     print 'the volume is:', s ** 3., '(cubic units)'

def cirsph():
     r = float(raw_input('enter length of radius: '))
     print 'the area is:', math.pi * (r ** 2.),
     '(units squared)'
     print 'the volume is:', (4. / 3.) * math.pi * (r **
3.), '(cubic units)'

sqcube()
cirsph()
```

**Q:**

Modulus. (a) Using loops and numeric operators, output all even numbers from 0 to 20

**A:**

11.

a.

```
for i in range(0, 22, 2):       # range(0, 21, 2) okay too
    print i
```

OR

```
for i in range(22):             # range(21) okay too
    if i % 2 == 0: print i
```

b.

```
for i in range(1, 20, 2):       # range(1, 21, 2) okay too
    print i
```

OR

```
for i in range(20):             # range(21) okay too
    if i % 2 != 0: print i
```

c.

when i % 2 is 0, it's even (divisible by 2), otherwise it's odd

## Chapter 6

**1:**

Strings. Are there any string methods or functions in the string module that will help me determine if a string is part of a larger string

**A:**

1.

```
find(), rfind(), index(), rindex()
```

**2:**

String Identifiers. Modify the idcheck.py script in Example 6-1 such that it will determine the validity of identifiers of length 1 as well as be able to detect if an identifier is a keyword. For the latter part of the exercise, you may use the keyword module (specifically the keyword.kwlist list) to aid in your cause

**A:**

2.

```
import string

alphas = string.letters + '_'
alnums = alphas + string.digits

iden = raw_input('Identifier to check? ')

if len(iden) > 0:
     if iden[0] not in alphas:
         print "invalid: first char must be alphabetic"
     else:
     if len(iden) > 1:
         for eachChar in iden[1:]:
              if eachChar not in alnums:
                  print invalid: other chars must be alphanumeric
              break
         else:
          import keyword
          if iden not in keyword.kwlist:
              print 'ok'
          else:
             print 'invalid: keyword name'

else:
    print 'no identifier entered'
```

## Chapter 7

**Q:**

Creating Dictionaries. Given a pair of identically-sized lists, say, [1, 2, 3, …], and ['abc', 'def', 'ghi', …], process all that list data into a single dictionary that looks like: {1: 'abc', 2: 'def', 3: 'ghi', …}.

**A:**

4.

```
# assumes both list1 and list2 are of the same length
dict = {}
for i in range(len(list1)):
    dict[list1[i]] = list2[i]
```

There is a more clever solution using the map() built-in function.

**Q:**

Inverting Dictionaries. Take a dictionary as input and return one as output, but the values are now the keys and vice versa

**A:**

7.

```
list1 = oldDict.values()
list2 = oldDict.keys()
```

(See solution to problem 4 for the remainder of this solution.)

## Chapter 8

**Q:**

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

**A:**

3a.

```
range(10)
```

**Q:**

Prime Numbers. We presented some code in this chapter to determine a number's largest factor or if it is prime. Turn this code into a Boolean function called isprime() such that the input is a single value, and the result returned is 1 if the number is prime and 0 otherwise

**A:**

4.

```
def isprime(num):
    count = num / 2
```

```
    while count > 1:
        if num % count == 0: return 0
        count = count - 1
    return 1
```

## Chapter 9

**1:**

File Access. Prompt for a number N and file F, and display the first N lines of F

**A:**

2.

```
file = open(raw_input('enter file: '))
allLines = file.readlines()
file.close()
num = input('enter number of lines: ')
i = 0
while i < num:
    print allLines[i],
    i = i + 1
```

(HINT: 1., 2., 3. can be inspired with `outfile.py` on p. 38 and p. 245)

**2:**

Logging Results. Convert your calculator program (Exercise 5-6) to take input from the command-line, i.e.

**A:**

14b.

```
import sys

print "# of args", len(sys.argv)
print "args:", sys.argv
```

**3:**

Searching Files. Obtain a byte value (0-255) and a file name. Display the number of times that byte appears in the file

**A:**

18.

part of this comes from `creatext.py` on p.247

## Chapter 10

**1:**

Raising Exceptions. Which of the following can RAISE exceptions during program execution? Note that this question does not ask what may CAUSE exceptions

**A:**

1.

e)

**2:**

Raising Exceptions. Referring to the list in the problem above, which could raise exceptions while running within the interactive interpreter?

**A:**

2.

**`try-except`** monitors the **`try`** clause for exceptions and execution jumps to the matching **`except`** clause. However, the **`finally`** clause of a **`try-finally`** will be executed regardless of whether or not an exception occurred.

## Chapter 11

**Q:**

Default arguments. Update the sales tax script you created in Exercise 5-7 such that a sales tax rate is no longer required as input to the function. Create a default argument using your local tax rate if one is not passed in on invocation

**A:**

5.

```
def printf(string, *args):
     print string % args
```

## Chapter 13

**Q:**

Functions vs. Methods. What are the differences between functions and methods?

**A:**

2.

Methods are basically functions, but are tied to a specific class object type. They are defined as part of a class and are executed as part of an instance of that class.

**Q:**

Delegation. In our final comments regarding the capOpen class of Example 13.4 where we proved that our class wrote out the data successfully, we noted that we could use either capOpen() or open() to read the file text. Why? Would anything change if we used one or the other?

**A:**

15.

It makes no difference whether we use `open()` or `capOpen()` to read our file because in `capOpen.py`, we delegated all of the reading functionality to the Python system defaults, meaning that no special action is ever taken on reads, meaning the same code would be executed, i.e., none of `read()`, `readline()`, or `readlines()` was overridden with any special functionality.

## Chapter 14

**1:**

Callable Objects. Name Python's callable objects

**A:**

1.

functions, methods, classes, callable class instances

**2:**

input vs. raw.input(). What is the difference between the built-in functions input() and raw_input()

**A:**

3.

`raw_input()` returns user input as a string; `input()` returns the evaluation of the user input as a Python expression.

## Chapter 15

**1:**

Recognize the following strings: bat, bit, but, hat, hit, or hut.

**A:**

1.

bat, hat, bit, etc.

```
[bh] [aiu] t
```

**2:**

Match any pair of words separated by a single space, i.e., first and last names

**A:**

2.

first name last

```
[A-Za-z-]+ [A-Za-z-]+
```

(any pair of words separated by a single space, e.g., first and last names, hyphens allowed)

**3:**

Match any word and single letter separated by a comma and single space, as in last name, first initial

**A:**

3.

last name, first

```
[A-Za-z-]+, [A-Za-z]
```

(any word and single letter separated by a comma and single space, e.g., last name, first initial)

```
[A-Za-z-]+, [A-Za-z-]+
```

(any pair of words separated by a comma and single space, e.g., last, first names, hyphens allowed)

**4:**

Match the set of the string representations of all Python longs

**A:**

8.

Python longs

```
\d+[lL]
```

(decimal [base 10] integers only)

**5:**

Match the set of the string representations of all Python floats

**A:**

9.

Python floats

```
[0-9]+(\.[0-9]*)?
```

(describes a simple floating point number, that is, any number of digits followed optionally by a single decimal point and zero or more numeric digits, as in "0.004," "2," "75.," etc.)

## Chapter 16

**1:**

**A:**

3.

TCP

**2:**

**A:**

5.

```
>>> import socket
>>> socket.getservbyname('daytime', 'udp')
13
```

# Other Reading and References

## Printed References

Altom,Tim,Mitch Chapman,*Programming with Python,* Prima,
Beazley,David M.,*Python Essential Reference,* New Riders,
Brown,Martin C.,*Python Annotated Archives,* McGraw Hill,
Grayson,John E.,*Python and Tkinter Programming,* Manning,
Hammond,MarkAndy Robinson,*Python Programming on Win32,* O'Reilly,
Harms,Daryl,Kenneth McDonald,*The Quick Python Book,* Manning,
Himstedt,Tobias,Klaus Mänzel,*Mit Python programmieren (Programming with Python)* [in German], dpunkt.verlag,
Lundh,Fredrik,(the eff-bot guide to) *The Standard Python Library,* http://FatBrain.com, (Product#)
Lutz,Mark,David Ascher,*Learning Python,* O'Reilly,
Lutz,Mark,*Programming Python,* O'Reilly,
Lutz,Mark,*Python Pocket Reference,* O'Reilly,
McGrath,Sean,*XML Processing with Python,* Prentice Hall,
Van Laningham,Ivan,*Teach Yourself Python in 24 Hours,* Sams,
Watters,Aaron,Guido van Rossum,James C. Ahlstrom,*Internet Programming with Python,* Henry Holt & Co./M&T Books/MIS:Press/IDG Books, [out-of-print]
van Rossum,Guido,*Python Library Reference:* Release 1.5.2, http://iUniverse.com,
van Rossum,Guido,*Python Reference Manual:* Release 1.5.2, http://iUniverse.com,
van Rossum,Guido,*Python Tutorial:* Release 1.5.2, http://iUniverse.com,
von Löwis,Martin,Nils Fischbeck,*Das Python-Buch (The Python Book)* [in German], Addison Wesley Longman, [out-of-print]

# Other Printed References

Aho,Alfred V.,Ravi Sethi,Jeffrey D. Ullman,*Compilers: Principles, Techniques, and Tools,* Addison Wesley Longman,
Brookshear,J. Glenn,*Computer Science, An Overview,* 6th Ed., Addison Wesley Longman,
Eckel,Bruce,*Thinking in C++,* 2nd Ed., Prentice Hall,
Eckel,Bruce,*Thinking in Java,* Prentice Hall,
Freidl,Jeffrey,*Mastering Regular Expressions,* O'Reilly,
Galvin,Peter,Abraham Silberschatz,*Operating System Concepts,* 5th Ed., Addison Wesley Longman,
McKusick,Marshall Kirk,Keith Bostic,Michael J. Karels,John S. Quarterman,*Design and Implementation of the 4.4BSD Operating System,* Addison Wesley Longman,
Ousterhout,John,*Tcl and Tk Toolkit,* Addison Wesley Longman,
Tane nbaum,Andrew S.,*Operating Systems: Design and Implementation,* Prentice Hall,
Welch,Brent,*Practical Programming in Tcl and Tk,* 3rd Ed., Prentice Hall,

## Online References

The list below represents a good number of Python online references. For a more up-to-date copy, check out the Python hotlist on the CD-ROM. Or better yet, go to the Core Python Programming website:

http://starship.python.net/crew/wesc/cpp

### Apache Modules

**mod_python** (evolved from *httpdapy* and *nsapy*)

http://www.modpython.org/

### Code

**Python source** (SourceForge)

http://sourceforge.net/project/?group_id=5470

**Snippet**s (SourceForge)

http://sourceforge.net/snippet/browse.php?by=lang&lang=6

SourceForge Python projects

http://sourceforge.net/search/?type_of_search=soft&words=python

**Vaults of Parnassus** (Python shareware)

http://www.vex.net/parnassus/

### Commercial

ActiveState Tool

http://www.activestate.com/

CyberWeb Consulting

http://www.roadkill.com/~wesc/cyberweb

O'Reilly Python DevCenter

http://www.oreillynet.com/python/

**PythonLabs** (commercial home page)

http://www.pythonlabs.com/

PythonWare

http://www.pythonware.com

ReportLab

http://www.reportlab.com/

UC Santa Cruz Extension Scripting Language Courses

http://www.ucsc-extension.edu/to/software/silang.html

## Communication

Bay Area Python Interest Group

http://www.baypiggies.org/

`comp.lang.python` weekly newsgroup summaries

http://purl.org/thecliff/python/url.html

Python Conferences

http://www.python.org/workshops/

**Python links** (sizeable hotlist)

http://www.cetus-links.de/oo_python.html

Python mailing lists

http://www.python.org/mailman/listinfo

Python Special Interest Groups

http://www.python.org/sigs

Starship FAQ

http://starship.python.net/~tbryan/FAQ/Starship/

Starship Python

http://starship.python.net

## Core

JPython

http://www.jpython.org/

**Python.org** (community home page)

http://www.python.org/

## DBs

Databases

http://www.python.org/topics/database/

Python database modules

http://www.python.org/topics/database/modules.html

Python DB API 2.0 specification

http://www.python.org/topics/database/DatabaseAPI-2.0.html">

## Extending

Extending and Embedding reference

http://www.python.org/doc/current/ext/ext.html

Python-C API

http://www.python.org/doc/current/api/api.html

**SWIG** (Simple Wrapper and Interface Generator)

http://www.swig.org

## GUIs (with Python development interfaces)

Gimp-Python

http://www.daa.com.au/~james/pygimp

**Glade** (GTK+ UI builder)

http://glade.pn.org

**GLC** (Glade Python Code Generator)

http://glc.sourceforge.net

**GTK+** (GIMP Toolkit)

http://www.gtk.org

**KDE** (K Desktop Environment)

http://www.kde.org

**PMW** (Python MegaWidgets for Tkinter)

http://www.dscpl.com.au/pmw/

**PyG Tools** (PyGTK, PyGNOME, etc.)

http://www.bioinformatics.org/pygtools

PyGTK Module

http://www.daa.com.au/~james/pygtk

PyQt-PyKDE

http://www.thekompany.com/projects/pykde

Python-KDE Tutorial

http://www.xs4all.nl/~bsarempt/python/tutorial.html

**Tkinter** (Python-Tk)

http://www.python.org/topics/tkinter

Tkinter intro (F. Lundh)

http://www.pythonware.com/library/tkinter/introduction

TrollTech Qt products (commercial)

http://www.trolltech.com/products

wxPython

http://www.wxpython.org/

## Macintosh

Macintosh Library Modules

http://www.python.org/doc/current/mac/mac.html

MacPython

http://www.cwi.nl/~jack/macpython.html

MacPython download page

http://www.python.org/download/download_mac.html

Open Directory MacPython links

http://dmoz.org/computers/systems/macintosh/development/languages/python

## News

Python Events

http://www.python.org/Events.html

Python mailing lists

http://www.python.org/mailman/listinfo

Python News

http://www.python.org/News.html

## Numerical/Scientific Processing

NumPy numerical extensions

http://www.python.org/topics/scicomp/numpy.html

**NumPy source**(SourceForge)

http://sourceforge.net/project/?group_id=1369

## Programming

Comparing Python to …

http://www.python.org/doc/Comparisons.html

Computer Programming for Everybody (CP4E)

http://www.python.org/cp4e/

CP4E proposal paper

http://www.python.org/doc/essays/cp4e.html

Empirical language comparison paper

http://wwwipd.ira.uka.de/~prechelt/Biblio/jccpprtTR.pdf

Guido's CP4E talk

http://www.python.org/doc/essays/ppt/acm-cp4e/

Instant Hacking: Learning to Program with Python

http://www.idi.ntnu.no/~mlh/python/programming.html

**Instant Python** (crash course in Python)

http://www.idi.ntnu.no/~mlh/python/instant.html

Learning to Program

http://members.xoom.com/alan_gauld/tutor/tutindex.htm

Non-Programmers Tutorial

http://www.honors.montana.edu/~jjc/easytut/easytut/

## Reference

Python documentation

http://www.python.org/doc

**FAQ** (Frequently Asked Questions)

http://www.python.org/doc/FAQ.html

FAQTS Python Knowledge Base

http://python.faqts.com

Global Module Index for the Python Standard Library

http://www.python.org/doc/current/modindex.html

Language Reference Manual

http://www.python.org/doc/current/ref/ref.html

Library Reference

http://www.python.org/doc/current/lib/lib.html

Python-Perl Cookbook

http://starship.python.net/crew/da/jak/cookbook.html

Quick Reference Guide

http://starship.python.net/quick-ref1_52.html

Regular Expressions HOWTO

http://www.python.org/doc/howto/regex/regex.html

## Releases

Python 1.5 to 1.5.2

http://www.python.org/1.5

Python 1.6

http://www.python.org/1.6

Python 2.0

http://www.pythonlabs.com/products/python2.0

Python Download

http://www.python.org/download

Python FTP site

ftp://ftp.python.org

What's New in 2.0

http://starship.python.net/crew/amk/python/writing/new-python

## Unicode

Python Unicode Integration (M.A. Lemburg)

http://starship.python.net/crew/lemburg/unicode-proposal.txt

Python Unicode Tutorial

http://www.reportlab.com/i18n/python_unicode_tutorial.html

Unicode Standard home page

http://www.unicode.org/

### Web

Five Minutes to a Python CGI (D. Mertz, Web Review)

http://webreview.com/pub/2000/07/07/feature/index02.html

HTMLgen home page

http://starship.python.net/crew/friedrich/HTMLgen/html/

Web programming

http://www.python.org/topics/web/

Writing CGI Programs in Python (P. Landers, Dev Shed)

http://www.devshed.com/Server_Side/Python/CGI/print.html

**XML**

Annotated XML 1.0 Specification

http://www.xml.com/axml/axmlintro.html

Python-XML How-To

http://www.python.org/doc/howto/xml/

Python-XML reference

http://www.python.org/doc/howto/xml-ref/

XML

http://www.python.org/topics/xml/

**XML** Cover Pages

http://www.oasis-open.org/cover/

XML FAQ

http://www.ucc.ie/xml/

Zope

http://www.zope.org/

## Python Operator Summary

TableC.1 represents the complete set of Python operators and to which standard types they apply. The operators are sorted from highest-to-lowest precedence, with those sharing the same shaded group having the same priority.

**Table c.1. Python Operators († - unary)**

| Operator | Int | long | float | complex | string | list | tuple | dictionary |
|---|---|---|---|---|---|---|---|---|
| [] | | | | | • | • | • | |
| [:] | | | | | • | • | • | |
| ** | • | • | • | • | | | | |
| +† | • | • | • | • | | | | |
| -† | • | • | • | • | | | | |
| ~† | • | • | | | | | | |
| * | • | • | • | • | • | • | • | |
| / | • | • | • | • | | | | |
| % | • | • | • | • | • | | | |
| + | • | • | • | • | • | • | • | |
| - | • | • | • | • | | | | |
| << | • | • | | | | | | |
| >> | • | • | | | | | | |
| & | • | • | | | | | | |
| ^ | • | • | | | | | | |
| \| | • | • | | | | | | |
| < | • | • | • | • | • | • | • | • |
| > | • | • | • | • | • | • | • | • |
| <= | • | • | • | • | • | • | • | • |
| >= | • | • | • | • | • | • | • | • |
| == | • | • | • | • | • | • | • | • |
| != | • | • | • | • | • | • | • | • |
| <> | • | • | • | • | • | • | • | • |
| is | • | • | • | • | • | • | • | • |
| is not | • | • | • | • | • | • | • | • |
| in | | | | | • | • | • | |
| not in | | | | | • | • | • | |
| not† | • | • | • | • | • | • | • | • |
| and | • | • | • | • | • | • | • | • |
| or | • | • | • | • | • | • | • | • |

# What's New in Python 2.0?

## Introduction

During the creation process of this text, the Python development team has been hard at work producing Python 2.0, which at press time, is finally being released to the public.

The supplementary CD-ROM in the back of the book contains the three most current releases of Python: 1.5.2, 1.6, and 2.0, including the most recent Java version of the Python interpreter, JPython (a.k.a. Jython): 1.1.

1.5.2 has been a rock stable release for almost two years, and is the foundation for most of the content in the book. 1.6 brought many new changes to Python. String methods and Unicode support have been added, as well as an improved regular expression engine.

A few more significant changes have made their way into the 2.0 release, which we will address here. We also recommend the "What's New in 2.0" document found at the Python 2.0 Web site (see the Online Resources section of the Appendix for the URL).

## Review and Preview

The following is a review of the major changes from Python 1.5.2 to 1.6, along with the expected bug fixes and module updates (new, revised, and obsoleted modules).

Unicode support

String methods

Upgraded regular expression engine (performance and Unicode enhancements)

New function invocation mechanism

The 2.0 release also features the usual fixes and module updates, but in addition, offers the following new features to the language:

Augmented assignment

List comprehensions

Extended `import` statement

Extended `print` statement

Once you get Python 2.0 compiled and/or installed on your system, you will see the familiar start-up line in UNIX (or similar if you are using another platform):

```
% python
Python 2.0 (#4, Oct  2 2000, 23:58:52)
[GCC 2.95.1 19990816 (release)] on sunos5
Type "copyright", "credits" or "license" for more
information.
>>>
```

Now, let's take a look at some of those new features.

## Augmented Assignment

Augmented assignment refers to the use of operators, which imply both an arithmetic operation as well as an assignment. You will recognize the following symbols if you are a C, C++, or Java programmer:

```
+=          -=          *=          /=          %=          **=
<<=         >>=             &=          ^=          |=
```

For example, the shorter

```
A += B
```

can now be used instead of

```
A = A + B
```

Other than the obvious syntactical change, the most significant difference is that the first object (`A` in our example) is examined only once. Mutable objects will be modified in place, whereas immutable objects will have the same effect as "`A = A + B`" (with a new object allocated) except that `A` is only evaluated once, as we have mentioned before.

```
>>> m = 12
>>> m %= 7
>>> m
5
>>> m **= 2
>>> m
25
>>> aList = [123, 'xyz']
>>> aList += [45.6e7]
>>> aList
```

```
[123, 'xyz', 456000000.0]
```

These in-place operators have equivalent special method names when creating classes to emulate numeric types. To implement an in-place special method, just add an "i" in front of the not-in-place operator; e.g., implement `__iadd__()` for the `+=` operator as opposed to `__add__()` for just the `+` operator.

## List Comprehensions

Remember how we used **`lambda`** functions along with `map()` and `filter()` to apply an operation to list members or to filter out list members based on criteria via a conditional expression? Well, list comprehensions simplify that task and improve performance by bypassing the necessity of using **`lambda`** along with functional programming built-in functions. List comprehensions allow you to provide an operation directly with an iteration over the original list sequence.

Let's take a look at the simpler list comprehension syntax first:

```
[ expression
                                        for
                                        iterative_var
                                        in
                                        sequence ]
```

The core of this statement is the **`for`** loop, which iterates over each item of *`sequence`*. The prefixed *`expression`* is applied for each member of the sequence, and the resulting values comprise the list that the expression yields. The iteration variable need not be part of the expression.

Recall the following code seen earlier in the text (Chapter 11, Functions) which utilizes a **`lambda`** function to square the members of a sequence:

```
>>> map(( lambda x: x ** 2), range(6))
[0, 1, 4, 9, 16, 25]
```

We can replace this code with the following list comprehension statement:

```
>>> [ x ** 2 for x in range(6) ]
[0, 1, 4, 9, 16, 25]
```

In the new statement, only one function call (`range()`) is made (as opposed to three—`range(), map(),` and the **`lambda`** function). You may also use parentheses around the expression if "`[ (x ** 2) for x in range(6) ]`" is easier for you to read. This syntax for list comprehensions can be a substitute for and is more efficient than using the `map()` built-in function along with **`lambda.`**

List comprehensions also support an extended syntax with the **`if`** statement:

```
[ expression
                                                    for
                                                    iterative_var
                                                    in
                                                    sequence
                                                    if
                                                    cond_expression]
```

This syntax will filter or "capture" sequence members only if they meet the condition provided for in the *`cond_expression`* conditional expression during iteration.

Recall the following `odd()` function below, which determines whether a numeric argument is odd or even (returning 1 for odd numbers and 0 for even numbers):

```
def odd(n):
    return n % 2
```

We were able to take the core operation from this function, and use it with `filter()` and **`lambda`** to obtain the set of odd numbers from a sequence:

```
>>> seq = [11, 10, 9, 9, 10, 10, 9, 8, 23, 9, 7, 18, 12, 11, 12]
>>> filter(lambda x: x % 2, seq)
[11, 9, 9, 9, 23, 9, 7, 11]
```

As in the previous example, we can bypass the use of `filter()` and **`lambda`** to obtain the desired set of numbers with list comprehensions:

```
>>> [ x for x in seq if x % 2 ]
[11, 9, 9, 9, 23, 9, 7, 11]
```

List comprehensions also support multiple nested **for** loops and more than one **if** clause. Please see the documentation including the "What's New" online document for more information.

## Extended **import** Statement

Another fairly common request from Python programmers is the ability to import modules and module attributes into your program using names other than their original given names. One common workaround is to assign the module name to a variable:

```
>>> import longmodulename
>>> short = longmodulename
>>> del longmodulename
```

In the example above, rather than using `longmodulename.attribute`, you would use the `short.attribute` to access the same object. (A similar analogy can be made with importing module attributes using **from-import**… see below.) However, to do this over and over again, and in multiple modules can be annoying and seem wasteful. The new extended **import** statement will now support the following:

```
>>> import longmodulename as short
```

Accordingly, you may also use this syntax with **from-import** statements.

```
>>> from sys import stderr as err
>>> err.write("now using sys.stderr")
now using sys.stderr
```

We will note that as is not a new keyword and is only recognized when using **import**. As a result, you can still use it as a valid identifier in your code:

```
>>> as = 14
>>> as += 3
>>> as
17
```

## Extended **print** Statement

One of the last and more argumentative additions to Python 2.0 is the extended **print** statement. The change, which employs a pair of "greater than" symbols (>>), allows you to direct the output of **print** to a file other than standard output.

In the example below, we utilize our import of `sys.stderr` to `err` above:

```
>>> print >> err, "using sys.stderr again"
using sys.stderr again
```

## Conclusion

While both augmented assignments and list comprehensions appear to be adding a twist to Python's easygoing syntax, the basic philosophy of keeping the language clean and simple has not changed. The key value-adds that these new features bring to the table is actually under the covers.

Augmented assignment only evaluates the first object once—a timesaver and performance enhancer over the long haul. Also, because function objects created by **lambda** are practically the same as those generated by **def**, the overhead of a real function call is incurred when they are executed. By using list comprehensions, there is no additional function object created on the fly, nor is there the additional function call overhead present. In this sense, list comprehensions give Python more "inlined" execution.

The extended **import** and **print** statements have less to do with performance as they do with programmer convenience.

Other additions include an optional garbage collector that can detect cycles and improved XML support (`xml.dom`, `xml.sax`, `xml.parsers`, and `pyexpat` modules). Other features to look for in 2.0 are range displays, parallel **for** loops, and ports to 64-bit systems.

## Exercise

**Q:**

Create a composite list comprehension statement that creates (randomly) a list of between 1 and 10 random numbers, ranging from 1 to 100, and pull out only the odd ones.

**A:**

*Answer:*

Our solution uses list comprehensions as well as the new extended **import** syntax.

```
>>> from random import randint as ri
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[47, 9, 85]
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[45, 3]
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[]
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[47, 25, 95, 83, 15, 77]
```